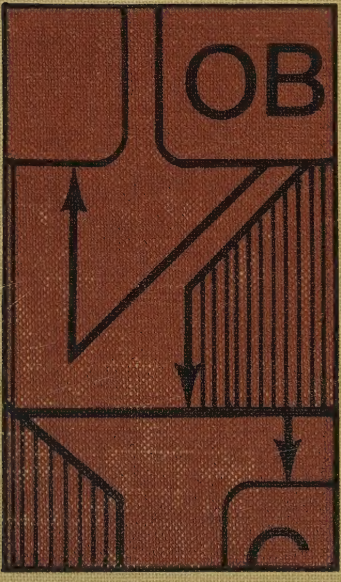
Automatic Transaction Decomposition in a Distributed CODASYL Prototype System



UMI Research Press

Computer Science: Distributed Database Systems

Automatic Transaction Decomposition in a Distributed CODASYL Prototype System

Computer Science: Distributed Database Systems, No. 6

Harold S. Stone, Series Editor

President, The Interfactor, Inc.

Other Titles in This Series

No. 1 *Medusa: A Distributed Operating System*	John K. Ousterhout
No. 2 *Dynamic Resource Allocation in Distributed Computing Systems*	Charles R. Vick
No. 3 *Performance Evaluation of Data Base Systems*	Patrick M.K. Wong
No. 4 *A Distributed Double-Loop Computer Network (DDLCN)*	Jacob J. Wolf, III
No. 5 *Performance of Update Algorithms for Replicated Data*	Hector Garcia-Molina
No. 7 *Data Management on Distributed Databases*	Benjamin Wan-San Wah
No. 8 *Update Synchronization in Multiaccess Systems*	Milan Milenkovic
No. 9 *Data Security and Performance Overhead in a Distributed Architecture System*	John M. Cary
No. 10 *Efficiency Analysis of File Organization and Information Retrieval*	J. Dennis Omlor
No. 11 *Search Mechanisms for Large Files*	Marie-Anne Kamal Neimat
No. 12 *Optimization of Queries in Relational Databases*	Yehoshua Chaim Sagiv

Automatic Transaction Decomposition in a Distributed CODASYL Prototype System

QA
76.9
D5
G46
1981

by
Frank Germano, Jr.

UMI RESEARCH PRESS
Ann Arbor, Michigan

Produced and distributed by
UMI Research Press
an imprint of
University Microfilms International
Ann Arbor, Michigan 48106

Library of Congress Cataloging in Publication Data

Germano, Frank.
Automatic transaction decomposition in a distributed CODASYL prototype system.

(Computer science. Distributed database systems ; no. 6)
Revision of thesis (Ph.D.)–University of Pennsylvania, 1980.
Bibliography: p.
Includes index.
1. Electronic data processing–Distributed processing.
2. Data base management. I. Title. II. Series.

QA76.9.D5G46 1981 001.64'25 81-12943
ISBN 0-8357-1221-4 AACR2

TO

J. K. G.

12-21-74

Contents

List of Figures

Acknowledgments

I would like to thank Peter Buneman, Eric Clemons, Rob Gerritsen, Howard Morgan, and Noah Prywes, for their time and effort in reviewing several drafts of this project. I am particularly indebted to Rob Gerritsen, whose encouragement and guidance have meant a lot to me.

I am also indebted to the Decision Sciences Department and Howard Kunreuther for providing me a home away from home. The interdisciplinary interaction between Decision Sciences and Computer Science, which this arrangement has allowed, has been of great benefit to me. Discussions with Peter Buneman led me in the direction of high-level languages in the distributed environment. Eric Clemons and I have had several interactions involving multiple schema architectures and external schema facilities for CODASYL systems.

Partial support by ONR Grant N00014-78-G0302 is gratefully acknowledged.

I would also like to thank Meru Thakur for his help with the preprocessor and Laurine States for her help with an early version of the model development.

Lastly, I would like to thank my colleagues and friends at the University of Pennsylvania, Temple University, and Digital Equipment Corporation for their support.

1

Introduction

1.0 Introduction

Current hardware cost trends and associated trends in the communications and networking fields make the distribution of a database ever more attractive. The key to the satisfactory distribution of a database is the existence of Distributed Database Management Systems (DDBMS) software to maintain the distributed data structure. The research reported here deals with the design and implementation of DDBMS's, which support CODASYL-like structures in a geographically transparent fashion, freeing the application's programmer from concern with data location. A global schema data definition language is defined to provide this geographic transparency. A high-level FOR EACH data language is used to represent transactions. These high-level transaction descriptions are decomposed into associated front-end and back-end tasks with all details of interprocess communications handled by the DDBMS. Four possible software architectures are considered and their optimal performance characteristics studied with 0–1 integer, nonlinear optimization models.

1.1 Contribution

This project makes several related contributions to the understanding of the implementation of Distributed Database Management System software.

1.1.1 Automatic Transaction Decomposition

The major contribution of this study is the process used to automatically decompose a transaction into a control task and subordinate support tasks. A DDBMS utility translates the user transactions into a front-end control task and associated back-end support tasks. A *front-end* task interfaces directly to the application program and is part of its process; a *back-end* task interfaces directly to a database structure at a given host. Front-end and back-end tasks communicate using network services connecting the hosts. These communication details are handled by the DDBMS decomposition utility and not the user. This decomposition process, coupled with a global view mechanism, provides geographic transparency for the user.

1.1.2 CODASYL Extensions For The Distributed Environment

Access-path (CODASYL-like) architectures are shown to be a viable basis for a distributed database management system, if the architectures are augmented by high-level language facilities and global view mechanisms. Use of high-level language facilities improves the performance of distributed systems.

It is not enough to extend CODASYL by using a set-level language, a language where one DML action processes a set of record instances rather than a single record instance. In these set-level languages, sets of records can be defined in several ways. A value for a CODASYL CALC key defines the membership of a *CALC-set* instance; an owner instance of a CODASYL set defines the membership of a *CODASYL-set* instance; and a record type and CODASYL area identification defines the membership of an *AREA-set*. Instead of set-level languages, program fragments whose processing steps are localized and which reference multiple record types must be used. These program fragments that reference multiple record types will be referred to as a *high-level* DML.

1.1.3 Definition of Prototype Architecture

A prototype architecture, *DSEED*, is defined to illustrate the interplay of the key software design decisions which go into a distributed database management system. Key features of this prototype architecture include: a high-level data manipulation language; a pre-processor to map the high-level language into a front-end control task and associated back-end tasks; an implementation for distributed sets, sets whose owner and members are not collocated; a global view mechanism and associated data definition language to provide geographic transparency; and stream communications. The use of a pre-processor and its associated task decomposition algorithm is a key contribution of this project.

1.1.4 Multi-Schema Architecture

A multi-schema architecture is proposed for the distributed environment which extends the ANSI/SPARC database recommendations [ANSI 76] to the distributed environment. The schema structure of the DSEED prototype is shown to be an example of this architecture. The utility of the multi-schema architecture in the heterogeneous data model environment is demonstrated.

1.1.5 Analytic Performance Models

Four analytic performance models of access-path architectures are developed and their relative performance compared for a sample database structure and transaction mix. Four alternative methods for distributing CODASYL systems were identified: the *remote area model*, where the areas as files are distributed and file-level commands are transmitted in the network; the *remote database model*, where whole databases are distributed as back-ends and DML-level commands are transmitted; the *high-level language model*, where back-end databases are used as before and high-level commands are transmitted; and the *high-level language model with parallel polling*, where the hosts with transaction entry-records are found by parallel polling rather than sequential scan. The operation of the four architectures was studied for a hypothetical, but realistic, order-parts database with six transaction classes.

1.2 Relevance

The distribution of CODASYL-like structures is addressed by this study. In almost all other cases, the RELATIONAL model has served as the basis for distributed database work [Bernstein 77, Epstein 78, Rothnie 77b, Schneider 77, Stonebraker 76, Stonebraker 77]. Our attention to CODASYL-like structures does not endorse the existing CODASYL standards [CODASYL 71, CODASYL 78], as changes must be made for these architectures to perform well in a distributed environment. Nor does it reduce the importance of the current distributed relational research conducted by others. Each architecture will have its optimal domain of applicability. The prototype architectures discussed here are geared to a business environment where transaction classes are identified, programmed, and installed beforehand, where data processing is somewhat localized, and where there is little concurrent access to the same objects. Much of actual data processing fits these assumptions.

1.2.1 Why Distribute the Data?

Data is distributed for both organizational and performance reasons. Many applications, because of their nature, involve the whole organization. These applications can span both organizational and geographic boundaries. For performance reasons, current technology has required that databases serving such environments be centralized. This centralization is in direct conflict with organizational pressures for local control. The advance of communications and networking technology, coupled with a recognition that large portions of a distributed database are typically referenced locally, has resulted in a desire to distribute the database. Both factors may lead to increased overall performance, although individual instances of performance degradation may occur.

1.2.2 Distributed Processing Vs. Distributed Databases

The term *distributed processing* has been broadly used by the data processing profession to refer to any computer activity involving processing at a location remote from the user. (The term *locality of reference* is commonly used to define the situation in distributed databases where local data are predominantely used locally.) Typically, the user employs a remote terminal or remote job-entry station to access data on a remote computer. This situation is summarized in figure 1–1.

Figure 1–1. Distributed Data Processing

```
                                         ----------
                                        |          |
USERi •---------------------------------|          |
                                        |          |
                                        |  DATA    |
                                         ----------
```

The term *distributed database* is sometimes used to refer to the distributed processing situation; however, its use generally implies something more. A distributed database involves a remote user who accesses a remote collection of data. The remote collection of data is distributed, i.e., portions of the data reside on different machines. Because there are data relationships between the dispersed elements, which are important to the user and consequently should be recognized by the software maintaining the data structure, the database structure must be considered as a single unit. Figure 1-2 summarizes this situation.

Moreover, there is a desire to view the structure as if it resided on a single machine, i.e., to hide the distribution from the application programmer. This *geographic transparency* is desirable because it simplifies application programming. It also allows system optimization of complex decisions.

Figure 1-2. Distributed Database

```
                  ------------------------------------------------
                  |                                              |
                  |                ----------                    |
USERi  • ---------|                |        |                    |
                  |                | DATA   |                    |
                  |                ----------                    |
                  |                 /       \                    |
                  |                /         \                   |
                  |               /           \                  |
                  |              /             \                 |
                  |        -----------      -----------          |
                  |        |         |      |         |          |
                  |        |  DATA   |      |  DATA   |          |
                  |        -----------      -----------          |
                  |                                              |
                  |                                              |
                  ------------------------------------------------
```

1.2.3 The Distributed Database Environment

In order to put solutions to the distributed database problem into their proper perspective, the physical characteristics of the distribution environment must be considered. The effective process-to-process communications rate between interconnected computers defines three regions of interest. Figure 1-3 describes this environment. In this figure, the boxes are computer hosts and the lines are computer-to-computer connections or *interconnects.* Three levels of host interconnect are recognized: bus networks, local networks, and non-local networks. The actual names of the regions are not important; however, the actual characteristics of the communications which define the regions are important.

The bus networks imply a very tightly coupled architecture. They are not true data bus architectures like the DEC UNIBUS, which could be used to define a fourth region, but they are closely related. The bus architecture allows computer separation of a few hundred feet and data rates of the order of 10 megabits per second. Bus networks are indicated by triple lines (###) on figure 1-3.

The local networks are limited to a few miles and typically are designed to serve a single geographic complex. Lower data rates of the order of 56 kilobits per second are typical. It is true that nationwide networks can be built with lines of these speeds, but the effective rates are much lower. Local networks are indicated by double lines (: : :) on figure 1-3.

The non-local networks are the slow networks. Data rates of 9.6 kilobits per second represent the upper limit for these networks. Telephone technology, and associated cost characteristics fix the upper limits for these networks. Increased bandwidth can be purchased; however, after a point unused capacity drives the cost per actual message sent to prohibitive levels. Non-local networks are indicated by single lines (. . .) on figure 1-3.

A typical large corporation would have local networks at its geographic complexes. These local networks would connect to non-local networks and

Figure 1-3. Distributed Database Environment

```
-------------------------------------------------------------
|                                                           |
|      ------------------               ---------            |
|      |        |        |              |       |            |
|      | HOST  ### HOST  |              | HOST  |            |
|      |        |        |              |       |            |
|      ------------------               ---------            |
|           .    :                            :              |
|           .    ::::::::::::::::::::::::::::::              |
|           .                                                |
|           .                                                |
|           .                                                |
|           .                                                |
|      ----------            -----------                     |
|      |        |            |          |                    |
|      |  HOST  |:::::::::::|  HOST    |                    |
|      |        |            |          |                    |
|      ----------            -----------                     |
|                                                           |
-------------------------------------------------------------

      LEGEND:
        ### -- high-speed channel (BUS)
        ::: -- medium-speed channel (LOCAL)
        ... -- low-speed channel (NON-LOCAL)
```

could also include bus sub-nets. When a database is distributed it is important to understand which network environments are involved, as a solution to the distributed database problem may not be applicable to all regions of the distributed environment. Our focus is principally on building distributed databases across the non-local and local regions. It is in these regions, in contrast to the bus region, where the DDBMS architecture is more critically influenced by the characteristics of the distributed database environment.

1.2.4 DDBMS Software Design Decisions

Distributed Database Management System (DDBMS) software is systems software which manages a distributed database. This software must:

1. maintain consistency and integrity;
2. provide acceptable performance;
3. provide global views and geographic transparency;
4. provide recovery mechanisms; and
5. be secure and private.

These goals can be achieved, with varying degrees of success, by a myriad of software designs. Each implementation of DDBMS software is characterized by how these design decisions are made. The major design decisions are discussed below.

1.2.4.1 How should objects be named and located? Do you distribute files, records, fields, or something else? Whatever is distributed must be uniquely named so that the system can refer to it. The naming information is part of the schema, which is managed by the DDBMS. The location function is managed by directory services, which are also managed by the DDBMS. As the degree of granularity increases, the number of objects to track increases, making the directory problem more difficult to solve quickly.

1.2.4.2 How should data redundancy be handled? The inclusion of redundant data (copies) can improve reliability because a damaged or unavailable piece of data can be retrieved from an alternate source. Through increased availability of data, a failure is averted. Retrieval performance can be improved because a local or close copy can be retrieved rather than a remote copy. However, update performance is degraded because several copies must be consistently updated. Clearly, consistency control, the process of keeping all copies consistent, is more difficult with redundant data.

1.2.4.3 What data model(s) should be used? Should a single data model form the basis of the architecture (a homogeneous data model environment) or should multiple data models (the heterogeneous data model environment) be used? Two data model classes are of interest: relational and access-path. The *access-path* class includes the CODASYL (network), hierarchical, and hybrid, e.g., IBM's IMS data models. Our interest in access-path data models does not restrict us to navigational data manipulation languages. It is generally agreed that high-level data access languages are required in the distributed environment. Use of underlying CODASYL-like structures does not necessarily imply navigation over communications channels.

1.2.4.4 How should geographic transparency be provided? How should a user view of the distributed database and its associated geographic transparency be provided? Should a *top-down* process be employed starting with a single master schema which is then divided and distributed? This process is the logical extension of the ANSI/X3/SPARC committee recommendations

[ANSI 76] to the distributed environment. A strong database administration function will be required to implement any top-down process.

Or should a *bottom-up* process be employed starting with existing local database structures which support a unified data view derived using an integrating schema facility? In fact, several alternative unified views can be provided, each supporting its own class of applications. This approach is essentially an online database restructuring problem.

Bottom-up approaches represent a more difficult problem when compared to the top-down approaches. The difficulty arises because the software must map existing—and perhaps not too compatible—component databases to the desired unified view. Bottom-up approaches have the advantage of requiring less centralized database administration. Only the organizations that wish to share information must agree on the content of the unified view; and since multiple views are possible, views can be tailored to specific applications. Top-down architectures will usually provide better overall performance characteristics. This performance advantage is derived from the centralized control inherent in the top-down architecture and the strict compatability between the global view and its supporting local views. The requirement in the bottom-up approaches of supporting multiple views, which may not be very compatible, contributes to the bottom-up performance disadvantage. However, because of organizational issues favoring evolutionary software development, bottom-up architectures may be the only viable alternative.

1.2.4.5 How should program tasks be decomposed? Because the data required to process a transaction in the general case are at multiple locations, program tasks supporting the transaction must be processed at multiple locations. A decomposition process is necessary to define the relevant subtasks to be performed at each location. This decomposition can be performed by the applications programmer, resulting in no geographic transparency, or it can be performed by the DDBMS. In this case, the application programmer writes programs, which are then decomposed by the DDBMS software into the necessary remote sub-tasks.

1.2.4.6 How should concurrent access be controlled? Because of the effective process-to-process communications rates, concurrent access control by traditional techniques is slow. Concurrency control is typically implemented by setting resource locks at some granularity level [Ries 79a]. Whereas the setting of a lock on a single host could occur in a 10 to 100 micro-second time range, the setting of a lock using inter-host, process-to-process communications occurs in a 10 to 100 milli-second range. The tightly coupled protocols of the single host no longer perform well in this new environment.

Nevertheless, a method must be chosen if a consistent data structure is to be maintained.

Whenever concurrent access to multiple resources is allowed, the possibility of a deadlock situation exists. The selected concurrency control algorithm must include provisions to handle deadlocks. The success of the concurrency control method chosen depends very much on the rate at which conflicts, either simultaneous access to objects or deadlocks, occur. The level of granularity also affects performance [Ries 79b].

1.3 Overview

The optimal (minimized operational cost) distribution of a data structure depends upon many factors, the most important being the nature of the data and how the data will be used. The implementor of DDBMS software must make many design decisions. Each piece of software so designed will have a defined optimum range of applicability where its performance, measured by all relevant criteria, surpasses alternative designs.

A prototype DDBMS, DSEED, will be described below. In order to place DSEED in the context of Distributed Database Management System software, the software design of DSEED is summarized relative to the software design decisions described earlier.

1.3.1 The Data Model of DSEED

Researchers have suggested that access-path architectures, e.g., CODASYL, are a poor choice for a basis data model in the distributed environment [Stonebraker 79a]. Since most production databases in existence today are built with access-path architectures for performance reasons, we decided to investigate the question: "Can access-path architectures be used as a basis for distributed work?" For these reasons, a CODASYL-like architecture was chosen as a basis data model for DSEED. Various units of distribution are considered. These include the database, the area, and the record. In DSEED, the decision was made to assign data record types to underlying databases.

In DSEED, the problem of finding a record's geographic location is solved by using data directories or parallel polling. Transaction entry-records are found using either parallel polling or directory searching. Data record instances along an access path are found using a distributed directory structure which augments the local CODASYL databases.

1.3.2 Geographic Transparency in DSEED

The achievement of geographic transparency was considered an important goal of a DDBMS. For this reason, a global schema mechanism using a top-down design process is included. A top-down process was chosen because it was believed that it represented a simpler problem.

1.3.3 Program Task Decomposition in DSEED

The automatic decomposition of a transaction into component front-end and back-end tasks contributes to providing geographic transparency. Allowing this operation to be accomplished by the system introduces the possibility of doing some system optimization.

1.3.4 Data Redundancy and Concurrency Control in DSEED

Although recognized as key problems in DDBMS implementation, data redundancy and concurrency control are not addressed at any length in the DSEED prototype. Because of the locality of reference assumption, discussed earlier, the availability of a data redundancy mechanism is less critical in the DSEED environment. Since the concurrency control issue is under active investigation by a number of researchers [Lin 79, Minoura 79, Menasce 78, Rosenkrantz 78, Rothnie 77a, Stonebraker 77], little time was spent addressing this problem area. However, the concurrency mechanism found in the MADTRAN/MADMAN system [Rosenkrantz 78] is compatible with the DSEED architecture and could be integrated into DSEED.

1.4 Summary and Overview of Following Chapters

In this chapter, we have attempted to place the work reported here into the context of DDBMS to support transaction processing. The schema facilities for this prototype architecture to support global views from supporting local views are defined in chapter 2. The syntax and semantics of the associated high-level data language, SEEDFE, are defined in chapter 3. In chapter 4, the overall structure of the DSEED prototype architecture is outlined. The use of a pre-processor and global schema information to decompose a SEEDFE program into remote tasks and the associated stream communications are defined in this chapter. In chapter 5, four possible DDBMS architectures based on access-path architectures are analyzed. Analytic performance models of these architectures are developed and used to demonstrate the superior performance, for a sample database, of the high-level language approaches. In chapter 6, various facets of distributed update control are discussed in relation to the DSEED architecture. The last chapter presents conclusions of the research effort and discusses future research areas.

2

Schema Definition Facilities

2.0 Introduction

A distributed database is a data collection whose components are distributed among several geographically dispersed locations. Moreover, data relationships between dispersed data elements exist which must be recognized and supported by the Distributed Database Management System (DDBMS) software. To support dynamically a global view spanning multiple databases, special global schema facilities must be defined. The requirements and possible syntax of such global schema facilities are discussed in this chapter.

In a multiple computer environment where each computer manages the local data structure, i.e., a remote DBMS architecture, a mechanism is required to support the data view derived from the dispersed data instances. This mechanism, which Jim Fry called an "Integrating Schema Facility" or "Aggregate Schema" [Fry 77, Fry 78], is a central component of a DDBMS. While Fry focused on an integrated view represented by a new database constructed from several existing, independent databases with batch restructuring techniques, this study focuses on a view materialized dynamically at execution time. But since it is known (by choice) which views will be required when the database is designed, additional information can be added to the data structure to support efficient dynamic view materialization.

In section 2.1 various aspects of distributed structures will be developed. First, the relationship of the multiple local structures which support a global structure in a distributed environment is defined. Based on the logical structure of the schemas, distributed structures are classified as homogeneous or heterogeneous. Next, the concept of a distributed set, a set where the owner instance and member instances may reside in different, geographically-dispersed databases, is defined along with supporting schema facilities. As an alternative to polling to find the starting record instance in a distributed environment, an entry-record directory facility is proposed. The section concludes with a discussion of sorted sets in a distributed environment.

In section 2.2 the operational environment of the Data Base Administrator (DBA) is discussed. The syntax and semantics of the various Data Definition Languages (DDL's) to support a DDBMS are defined. Specifically, a global internal schema DDL to represent the global internal schema is defined. The relationship between the global internal schema and the

Figure 2-1. Schemas in a Distributed Environment

```
EXTERNAL          EXTERNAL                EXTERNAL
SCHEMA 1          SCHEMA 2                SCHEMA n
    .                .                        .
    .                .                        .
--------------- External  Interface --------------------
                               .
                               .
              GLOBAL CONCEPTUAL SCHEMA
                               .
                               .
---------------- Conceptual Interface ------------------
                               .
                               .
               GLOBAL INTERNAL SCHEMA
                               .
                               .
-------------- Internal Interface ----------------------
 .                 .                             .
 .                 .                             .
LOCAL             LOCAL                         LOCAL
INTERFACE         INTERFACE                     INTERFACE
SCHEMA 1          SCHEMA 2                      SCHEMA m
 .                 .                             .
 .                 .                             .
LOCAL             LOCAL                         LOCAL
CONCEPTUAL        CONCEPTUAL                    CONCEPTUAL
SCHEMA 1          SCHEMA 2                      SCHEMA m
 .                 .                             .
 .                 .                             .
LOCAL             LOCAL                         LOCAL
INTERNAL          INTERNAL                      INTERNAL
SCHEMA 1          SCHEMA 2                      SCHEMA m
 .                 .                             .
 .                 .                             .
PHYSICAL          PHYSICAL                      PHYSICAL
DATABASE 1        DATABASE 2                    DATABASE m


  [NOTE: The local external, local conceptual, and local
          internal interfaces  are not shown.]
```

underlying local schemas is discussed to show how the global internal schema is efficiently supported by the underlying structures. The operation of the Distributed Database Schema Definition Processor (DDBSDP) is defined. Using simple heuristics this processor can automate many of the decisions made by the DBA. The section concludes with the DDL definitions for the examples given in the prior section.

2.1 Distributed Structures

2.1.1 Local and Global Schema Structures

The ANSI/SPARC study group proposed a three-schema model to support Database Management Systems [ANSI 76]. They proposed a set of external schemas to meet the needs of the application programmers, a single conceptual schema to define the information needs of the organization, and a single internal schema to describe the structure of the database, indices, and storage devices used.

In the distributed environment it is useful to replace the ANSI/SPARC concept of a single internal schema with a single *global* internal schema augmented by several sets of *local* schemas (interface, conceptual, and internal). Figure 2-1 diagrams this structure. A local internal schema will describe each physical database at a host; a global internal schema will describe the data view supported by the several physical databases. A global internal schema has some awareness of the geographic partition into supporting databases. Currently, in DSEED, all local interface schemas are sub-schemas of CODASYL schemas, all local internal schemas are CODASYL schemas, and local conceptual schemas do not exist.

The ANSI/SPARC concept of a conceptual schema is preserved but in a restricted form. The *global conceptual* schema does not represent the entire information needs of the organization. It represents the information needs of the applications served by the database and nothing more. However, the global conceptual schema does insulate the application programs from the details of data structure implementation. A global conceptual schema represents the logical data structure of the global internal schema, i.e., only records, sets, and items are known to it. A global conceptual schema is insensitive to the geographic location of individual supporting databases.

Although not strictly related to logical structure, the global conceptual schema does have knowledge about which items can be efficiently accessed by key. The argument can be made that this information belongs in the global internal schema. Nevertheless, since access paths are already present in our global conceptual schema and since knowledge of where keyed access is efficient strongly impacts performance, this information is placed in our global conceptual schema. However, the mechanics by which this efficient keyed access is accomplished are specified in this global internal schema.

Several interface mechanisms are required to yield the desired data independence. The individual user schemas (external schemas) are related to the global conceptual schema through the *external interface*. The global conceptual schema is related to the global internal schema by the *conceptual interface*. The global internal schema is related to the local interface schemas by the *internal interface*.

2.1.1.1 The internal interface. The internal interface must map between the global internal schema and the local interface schemas. All questions of redundant storage of data, data access to several physical databases using communication channels, and simultaneous access must be addressed by this interface.

2.1.1.2 The conceptual interface. The conceptual interface insulates the external schema functions from the physical characteristics of the data structure. The conceptual interface is particularly useful when considering a heterogeneous data model environment. A conceptual schema, represented by a single data model, can be supported by multiple, underlying schemas, represented by different data models. At present, in DSEED, the global conceptual interface is essentially a null interface; the global conceptual schema degenerates into the global internal schema. For this reason, all future references to global schemas are in terms of global internal schemas.

2.1.1.3 The external interface. Given the global schema, a new "virtual structure," described by the external schema, must be constructed via a suitable mapping operation. These mappings can be considerably more complex than simple record, set, and item subset selection from the global conceptual schema. The mappings used to support dynamic restructuring are more indicative of the power which can be built into the external interface [Gerritsen 76]. The external interface may also support high-level, non-procedural specification of the user program function, simplifying the application programming task [Clemons 76]. For the present, external schemas in DSEED are analogous to CODASYL sub-schemas of the global internal schema.

External schemas may represent a restricted view of the database. A *restricted* external schema represents a data view derived from a single location, i.e., an external schema data view derived through *geographic restriction.* Consider the example posed by a distributed order database. A data view restricted to a single marketing branch would be described by a restricted external schema. A *non-restricted* external schema represents a data view whose components derive from one or more databases. Knowing that only data from a single location will be required allows for more efficient processing. External schema restriction to a single host is defined by the DBA at schema definition time using Data Definition Language facilities or at execution time using DML facilities.

2.1.2 Schema Structure Classification

Distributed DBMS schema structures are classified according to whether the database distribution structure is homogeneous or heterogeneous. In a *homogeneous schema* the local structures are equivalent. In a *heterogeneous schema* the local structures are not equivalent. Local database structures are classified according to logical (conceptual) structure, i.e., without regard to implementation characteristics. Number of pages in an area, set implementation mode, or record location mode do not affect logical structure. A heterogeneous DDBMS can contain homogeneous components.

2.1.2.1 Homogeneous schema structures. Consider an order-entry database for a nationwide marketing organization. Suppose each marketing branch has its own order database. Assume that these databases have the same structure, i.e., a homogeneous environment. An internal schema, representative of the local and global schemas of a typical order-entry database, is illustrated in figure 2–2. In this example, the global schema is identical to the local schemas.

Figure 2–2. Order Entry D/B Local/Global Internal Schemas

```
                LOCAL/GLOBAL INTERNAL schemas


                                   -----------------
          Order Number             |               |
            CALC------------>      |   ORDER-DATA  |
                                   |               |
                                   -----------------
                                           .
                                           .  LINES
                                           V
                                   -----------------
                                   |               |
                                   |   LINE-ITEM   |
                                   |               |
                                   -----------------
```

2.1.2.2 Heterogeneous schema structures. Consider two databases: a parts database and a supplier database with the local internal schemas illustrated in figure 2–3.

Figure 2–3. Part-Supplier Local Internal Schemas

```
                    LOCAL INTERNAL SCHEMAS

                  PARTS DATABASE         SUPPLIER DATABASE

                  -------------          ------------
Part Number       |           |          |          |
CALC-------->     |   PART    |          | SUPPLIER |
                  |           |          |          |
                  -------------          ------------
                                               .
                                               . SUPPLIES
                                               V
                                         ------------
                                         |          |
                                         |  DETAIL  |
                                         | pn, qty  |
                                         ------------
```

The global internal schema, illustrated in figure 2–4, could be constructed to represent a consolidated view of the data. The PART, SUPPLIER, and DETAIL records, respectively, of the global internal schema of figure 2–4, are derived from the PART, SUPPLIER, and DETAIL records of the local schemas of figure 2–3. Additional information, defined in section 2.4, is added to the local internal schemas to support efficient mapping of the local data instances into the global structure. This information, represented by additional record and set types added to the local schema declarations, causes additional record instances to be included in the database.

2.1.3 Distributed CODASYL Sets

A heterogeneous DDBMS may contain CODASYL sets whose owner and member record instances reside at different host sites. These sets will be referred to as *distributed sets.* If the underlying PARTS and SUPPLIER databases supporting the global internal schema of figure 2–4 are not collocated, the SUPPLIED-BY set is a distributed set. Distributed sets illustrated by global internal schema diagrams will include a "d" along with the set name to indicate this fact.

Figure 2–4. Part-Supplier Global Internal Schema

```
                    GLOBAL INTERNAL SCHEMA


                   ----------             ------------
    Part Number    |        |             |          |
 CALC   ------->   |  PART  |             | SUPPLIER |
                   |        |             |          |
                   ----------             ------------
                         \                    /
                          \                  /
          SUPPLIED-BY      \                /   SUPPLIES
                            V              V
                            ---------------
                            |             |
                            |   DETAIL    |
                            |  pn, qty    |
                            ---------------
```

In many cases, a distributed set can contain member records from many locations. Consider the database structure, illustrated in figure 2–5, of a company with several marketing branches and centralized inventory control.

The ORDER-DATA and DETAIL records are stored locally at the respective marketing offices; the PART records are stored at a centralized warehouse location. Because member records of a PARTS-RQD set instance can reside at several different marketing office locations, the PARTS-RQD set is distributed.

2.1.3.1 Alternative implementations of distributed sets. If a distributed set has a chain implementation mode, the set members cannot be retrieved if one of the hosts holding member records goes down. For this reason, alternative implementations of record sets are preferable in a distributed environment.

2.1.3.2 Use of pointer arrays. A pointer array, stored with the set owner record, can be used to represent sets. Pointer arrays reduce the effect of a down host. A partial result can be obtained, excluding information from a down host. Update changes which affect the down host must be postponed until the down host returns. Pointer arrays are restricted to the use of physical pointers. A physical pointer includes host identification, database page, and record number on the page.

Figure 2-5. A Distributed Set

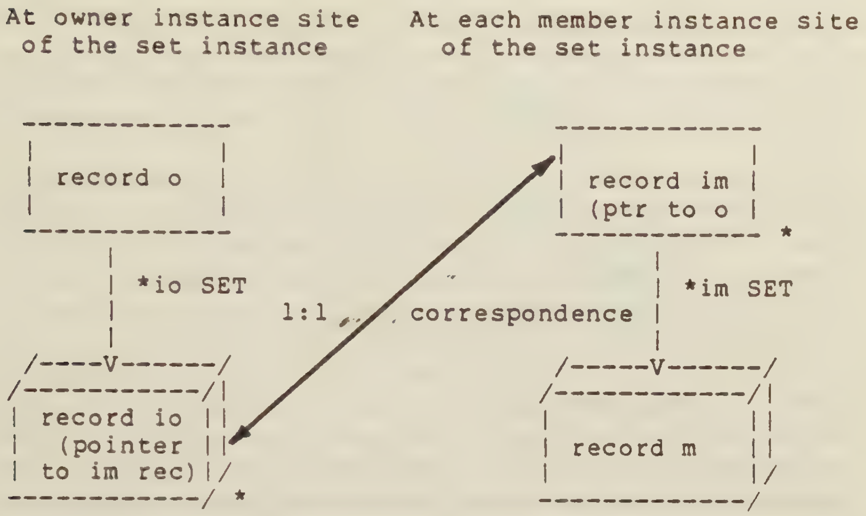
```
                GLOBAL INTERNAL SCHEMA

------------                      --------------
|          |                      |            |
|  PART    |                      | ORDER-DATA |
|          |                      |            |
------------                      --------------
          \                            /
           \                          /
PARTS-RQD   \ d                      /   LINES
             \                      /
              ----------------------
              |                    |
              | DETAIL part#       |
              |                    |
              ----------------------
```

2.1.3.3 Physical vs. symbolic pointers. Because the use of physical pointers restricts local database restructuring efforts, the use of symbolic pointers may offer an advantage. Symbolic pointers are composed of a sequence of key values which uniquely identify a data record instance. Consequently, record instances can be moved in the database and future accesses will still be able to locate the record. Unfortunately, use of symbolic pointers requires the storage of character strings, which may be longer than the physical pointer, and results in slower access time because of the time to convert the symbolic pointer to its associated database key. In order to use symbolic pointers, structures more general than pointer arrays must be defined.

2.1.3.4 Proposed implementation of distributed sets. A new construction is proposed to support distributed sets. It is currently defined in terms of additional record types which can automatically be generated by the schema definition processor. Defining additional record types allows us to "piggy-back" a distributed CODASYL implementation on top of existing DBMS implementations. Figure 2-6 summarizes this structure for distributed set j whose owner is of record type o and whose member is of type m.

Figure 2-6. Implementation of Distributed Set j


At owner instance site of the set instance
At each member instance site of the set instance
record o
record im (ptr to o
*
*io SET
1:1
correspondence
*im SET
record io (pointer to im rec)
*
record m

```
NOTE: A * indicates the additional record and
  set types added to the local schemas.
```

The record/set structures flagged by asterisks represent information added to the local data structures to support distributed sets. This information is not present in the global conceptual schema. The io record holds a pointer to im records, one for each host where records in the set j instance are found. Each instance of an im record is referred to as a *remote distributed set header.* The im record holds the pointer to the remote owner of the distributed set. The im record also serves to anchor the group of records of a set j instance which reside at the host. The set of records in a set j instance at a given host is called the *distributed set component* of set j at that host. The number of io record instances for a given instance of set j exactly matches the number of im record instances for that set. In other words, there is an io record instance for each host which has member records of the set j instance. This implementation of a distributed set forms the basis of a distributed directory. The io records indicate which hosts are involved in a specific distributed set instance and the im records indicate which specific records at each host.

2.1.4 Transaction Entry-Record Determination

In current CODASYL DBMS architectures, record instances to start transaction processing are determined by key value access (for one or several record instances) or a non-key value access (for all occurrences of a given record type). This class of records will be referred to as the *transaction entry-records.*

Records in a distributed system can be located in a static or dynamic fashion. The location of a record type is considered *static* when it is fixed at schema definition time; a record location is considered *dynamic* when it depends upon execution time characteristics. Record location is discussed more completely in section 5.2.4. In a distributed environment with dynamic record location, the host processing a key value entry-record transaction does not know where a record instance is located; consequently, all hosts which may have occurrences of the record type in question must be interrogated. This interrogation can occur sequentially by asking each relevant host and waiting for a response. Or, it can occur in parallel by polling. In this case, all relevant hosts are asked and then the relevant responses are collected in the order in which they return. The interrogation can be eliminated if an entry-record directory is maintained. However, directory maintenance will contribute to system overhead. With parallel polling the benefit of an entry-record directory may be negligible.

The maintenance of an entry-record directory can be requested by the DBA at schema definition time. Each entry-record type can have an entry-record directory to determine where specific entry-record instances are located.

Directory structures can be implemented in many ways. A basic directory structure is illustrated in figure 2–7. In this figure record type dh is the record instance entry. For a given record type and associated instance key value, exactly one dh record instance is reached. From this dh instance, de record instances can be retrieved. There is a de instance for each host which has the given record type and key value. The de entries only include host information; there is no indication of where that record is located at the given host. The actual location on disk of the record instance is determined at the local host.

The existence of an entry-record directory is not known by the global conceptual schema. The entry-point directory is part of the global internal schema.

2.1.5 Geographic Semantics

With transparent data distribution the applications programmer is not aware of data location. However, for some applications, the data location

Figure 2–7. Entry-Record Directory Structure

```
                              --------------------
                              | Record header dh |
   Rectype + Key Value --->   |                  |
                              |                  |
                              --------------------
                                       |
                                       |
                                       V
                              --------------------
                              |                  |
                              |   host pointer   |
                              |       de         |
                              --------------------
```

has semantics which are important to the application. Consider the nationwide order database of figure 2–2. The marketing vice-president would be interested in all the data, but sometimes segregated by marketing branch.

In order to recognize geographic structure, a geographic-based owner record could be included in the global conceptual schema. The global conceptual schema, included in figure 2–8, illustrates this technique. The

Figure 2–8. Geographic Semantics in the Order D/B

```
                    GLOBAL CONCEPTUAL SCHEMA

                                  --------------------
   Marketing Branch Number        |                  |
    KEYED---------------->        |   MARKET-BRANCH  |
                                  |                  |
                                  --------------------
                                           .
                                           .   ORDERS
                                           V
                                  --------------------
   Order Number                   |                  |
    KEYED---------------->        |   ORDER-DATA     |
                                  |                  |
                                  --------------------
                                           .
                                           .   LINES
                                           V
                                  --------------------
                                  |                  |
                                  |    LINE-ITEM     |
                                  |                  |
                                  --------------------
```

MARKET-BRANCH record has been added to support geographic processing. The MARKET-BRANCH record does not actually exist, but is synthesized by the DDBMS software.

Since order number might not be unique across individual marketing branches, accesses by order number may either have to be qualified by marketing branch or be prepared to receive multiple ORDER-DATA records. The choice of which situation to support must be made by the DBA at schema definition time through global conceptual DDL facilities. In a conceptual schema, efficient access to records by key is identified by the keyword "KEYED" instead of "CALC." The recent CODASYL specification [CODASYL 78] has made a similar change.

2.1.6 Sorted Sets in a Distributed Environment

The global internal schema may include sets which are sorted. If the PARTS-RQD set of figure 2–5 was sorted with the line number item of the DETAIL record serving as the sort key, we will usually require, for efficiency reasons, that the respective LINES sets of the local internal schemas be sorted on the same key. Dynamic construction of the global LINES set requires a merge of the component LINES sets by the internal interface.

Although other data restructuring operations, including sorting on an alternate key value, could be performed by the internal interface, they are not; alternatively, it is recommended that such restructuring be performed by the external interface. This simplifies the internal interface; only distributed issues, and not restructuring issues, need to be addressed.

2.2 Distributed DBMS Schema Management

2.2.1 The Distributed Schema Definition Processor

Using a top-down design philosophy the Distributed Database Schema Definition Processor (DDBSDP) will partially generate the local internal schema DDL files from the global internal schema. The DBA can modify these DDL files before requesting processing by the local schema processor. The DBA will primarily be concerned with changing local area size information, local set implementation modes, and local record location modes. With simple heuristics the DDBSDP can make the above decisions. Figure 2–9 summarizes the information flow of the schema definition process. With additional information, primarily frequency of occurrence, more efficient structures can be generated.

The global internal schema definition, GIDDL, is translated by the Distributed Database Schema Definition Processor, DDBSDP, into local

Figure 2-9. Schema Definition Processor Information Flow

```
                     heuristics
                          |
                          |
                          V
                     ----------
      GIDDL -----> | DDBSDP | -----> .GSH file
                     ----------        global internal
                      /   |      \      schema file
                     /    |       \
                    /     |        \
                   /      |         \
                  /       |          \
                 V        V           V
               LIDDL    LIDDL   ...  LIDDL
                 |        |           |
                 |        |           |
                 V        V           V
             -------  -------      -------
             | FDP |  | FDP |  ... | FDP |
             -------  -------      -------
                |        |            |
                |        |            |
                V        V            V
              .SCH     .SCH         .SCH
              local    local        local
              schema   schema       schema
              file     file         file
```

internal schema definitions, LIDDL, and an internal, compiled form. The local internal DDL definitions are then compiled by the local schema processors, FDP, to generate an internal, compiled form. The .GSH file is the internal, compiled form of the global internal schema. The .SCH files are the internal, compiled form of the local internal schemas.

The DDBSDP will generate the record definitions added to the local internal schemas to support distributed sets. These record types are the io and im record types described in section 2.1.3.4.

The DDBSDP will also generate the necessary record types to support entry-record directories. These record types are the directory header record (dh) and the key-record pair record (de) described in section 2.1.4.

2.2.2 DDBMS Data Definition Languages

The various Data Definition Languages required in the distributed environment are defined in this section. Specifically, the syntax and semantics of the global internal schema DDL is discussed. The syntax and semantics of the local internal schema DDL is typical of the 1971 CODASYL DBTG report [CODASYL 71] as implemented in the SEED DBMS [SEED 77]; consequently, only the use of the DDL to define the information added to the local schema is illustrated. Since the global conceptual schema was not addressed by the prototype implementation, it is not discussed. For this reason, the notion of "KEYED" access and geographic semantics have been included in the global internal DDL.

The syntax of the DDBMS global internal DDL for a remote database is defined in figure 2–10. The associated semantics are defined below.

The schema entry defines attributes of the schema itself. These attributes are inherited by the local internal schemas but may be changed by the DBA. Area size represents the total of all local area components. Without additional information the DDBSDP will make each local area component the same size.

The host-network control entry allows the DBA to define the internal interface between the DDBMS software and the telecommunications software. The primary function served is the enumeration of the hosts involved in supporting the distributed database for the global internal schema. The host name serves to link the schema to the host information table replicated at each host. This table contains such information as host communications id, cpu type, and local data model.

Many items used for symbolic pointers (owner or member) are a concatenation or calculation based on several items. Rather than require the explicit storage of such items, the rule to construct the item is defined in the virtual item clause. Using the current values of the component items the virtual item is materialized as required by the DDBMS.

The global record definition entry serves to define the source of the record. The host name qualification of the record name indicates whether the record always is derived from a single host or can be derived from several alternative host sites.

The location mode entry defines a mechanism to determine where physically to place a record instance, i.e., at what host and on what page. The decision of where to place record instances can be made at schema definition time or execution time. If all records of a given type are stored at a single host determined at execution time, the record location mode decision is *static*. If the record location mode decision depends upon execution time characteristics, the decision is *dynamic*. Two dynamic modes can be considered. The term *dynamic-local* will refer to the situation where all records originating at a site are stored locally at that site. The term *dynamic-mobile* will refer to the situation where a record's location is specified at execution time by the user or the system.

Figure 2-10. DDBMS Global Schema DDL

```
GLOBAL INTERNAL SCHEMA schema-name   [HOMOG | HETERO]
 [PRIVACY LOCK IS password]  DATABASE SIZE  IS i1 PAGES]
  [PAGE SIZE IS i2 WORDS]  [MAXIMUM OF i3 RECORDS PER PAGE].

DATABASE local-internal-schema-name
     HOSTS ARE [host-1] ...  .

AREA NAME IS a-name  [COMPONENTS AT [ ALL | host-1 ... ] ]
   [SIZE IS i4  PAGES ]   [PAGE IS i5 WORDS].


VIRTUAL v-i-name =  expression g-item-1, g-item-2, ... .


RECORD record-name
     [GEOGRAPHIC OWNER  ]
     [DERIVED FROM  [local-record-name [ AT host-name ] ]

LOCATION MODE
  [STATIC | DYNAMIC-LOCAL | DYNAMIC-MOBILE ]
  [KEYED USING global-item [DUPLICATES (LOCAL | GLOBAL) ]]
  [VIA global-set-name]

WITHIN area-name [REPLICATED AT host-name] .


global-item  {type declaration}
   [DIRECTORY GLOBAL AT host-name-1 ... [DUPLICATES ALLOWED].


SET global-set-name  [DISTRIBUTED]

OWNER     [SYSTEM]
   [ global-record-name
LINKED TO MEMBER USING  [global-or-virtual-item ]  ]]

MEMBER global-record-name
     [LINKED TO OWNER USING global-or-virtual-item  ]

ORDER IS FIRST|LAST|NEXT|PRIOR|SORTED]|

[ (ASCENDING | DESCENDING ) KEY global-or-virtual-item ]

DUPLICATES ARE
   (   FIRST | LAST    |  NOT ALLOWED  ) .
```

KEYED implies a desired access to the record by key equality and is usually implemented locally with a CALC location mode. The WITHIN entry defines the area and host for record storage. The host location is static, dynamic-local, or dynamic-mobile. Once stored at its primary site, the record instance is duplicated at the sites named in the REPLICATE entry.

Global items and local items are defined in a similar fashion with one exception: the maintenance of an entry-record directory (DIRECTORY GLOBAL) may be declared for the item.

The global set entry is qualified by "DISTRIBUTED" if the set is distributed. When a distributed set is sorted, duplicate record instances can be placed FIRST (ahead of the duplicate), LAST (behind the duplicate), or not allowed. The desire for a symbolic, rather than a physical, key can be specified in the LINK TO clauses of the OWNER and MEMBER clauses. As discussed earlier, symbolic keys offer greater flexibility in the distributed environment. Omission of the clauses in a distributed set definition forces the distributed schema parser to assume physical pointers. In this case, pointer fields of the distributed set implementation records contain database keys.

2.2.3 Schema Library Index

The support of distributed databases requires many additional schemas complicating the DBA's administrative task. To ease this burden in a production environment a schema library index, automatically maintained by the schema management software, is proposed. The schema library index is a data dictionary and is itself a distributed database. During prototype development the schemas were managed directly as files. A schema library index was not built.

2.3 Examples of DDL Definitions

2.3.1 Homogeneous Order Database Example

Figure 2–11 contains the global internal schema definition for the order database described in figures 2–2 and 2–8.

Using the top-down DDB schema definition processor, the local internal schemas will automatically be generated. Figure 2–12 illustrates the generated schemas. Because of the homogeneous nature of the global schema, the generated local schemas are very similar to the global schema. Without frequency information each local area is generated the same size. In this example each local area becomes 10 pages. The DBA can change this allocation by editing the generating DDL definition files prior to local FDP processing. If frequency of occurrence information is included, the DDPSDP processor can make a more realistic allocation.

Figure 2–11. Global Internal Schema for Order D/B

```
GLOBAL INTERNAL SCHEMA ORDERDB   HOMOG
     PRIVACY LOCK IS  "MKTVP"
     DATABASE SIZE IS  30 PAGES.

HOSTS ARE  BOSTON PHILA CHICAGO.

AREA NAME IS  ORDERF  COMPONENTS AT ALL.

RECORD MKT-BRANCH  GEOGRAPHIC HEADER.

RECORD ORDERDT
     LOCATION MODE DYNAMIC-LOCAL
     KEYED USING ORDERNO  DUPLICATES GLOBAL ALLOWED
     WITHIN ORDERF.
ORDERNO CHARACTER 5.
CUSTNO CHARACTER 5.
CUSTNAM CHARACTER 20.

RECORD LINITM
     LOCATION MODE DYNAMIC-LOCAL
     VIA LINES  WITHIN ORDERF.
LINENO FIXED.
OQTY FIXED.
OPN  CHARACTER 5.
OPRICE REAL.

SET LINES
     MODE IS CHAIN
     OWNER ORDRDT MEMBER LINITM
     ORDER IS SORTED  ASCENDING KEY LINENO  DUPLICATES NOT.

SET ORDERS
     MODE IS CHAIN
     OWNER MKT-BRANCH MEMBER ORDRDT.
```

2.3.2 Heterogeneous Supplier Database Example

The global internal schema for the supplier database of figure 2–4 is defined in figure 2–13.

The DDB Schema Definition Processor will generate the necessary local internal schemas. Because this heterogeneous database contains a distributed set, several additional records will be generated to support the dynamic materialization of the set. Figure 2–14 contains the schema diagrams, including generated records to support the distributed set, for the local internal schemas supporting the global internal schema. The generated global internal schemas are illustrated in figures 2–15 and 2–16.

The PART*IO and DETAIL*IM record types support the distributed set SUPPBY. There is a one-to-one relationship between instances of these

Figure 2-12. Local Internal Schemas for Order D/B

```
*HOSTID: BOSTON (similiar schemas for PHILA and CHICAGO)
SCHEMA ORDERDB
     PRIVACY LOCK IS "MKTVP"

AREA NAME IS ORDERF
     SIZE 10 PAGES.

RECORD ORDERDT
     LOCATION MODE CALC USING ORDERNO DUPLICATES NOT
     WITHIN ORDERF.
ORDERNO CHARACTER 5.
CUSTNO CHARACTER 5.
CUSTNAM CHARACTER 20.

RECORD LINITM
     LOCATION MODE VIA LINES  WITHIN ORDERF.
LINENO FIXED.
OQTY FIXED.
OPN  CHARACTER 5.
OPRICE REAL.


SET LINES
     MODE IS CHAIN
     OWNER ORDRDT MEMBER LINITM
     ORDER IS SORTED  ASCENDING KEY LINENO  DUPLICATES NOT.

SET ORDERS
     MODE IS CHAIN
     OWNER SYSTEM MEMBER ORDRDT
```

record types in the database. The PART*IO record contains two fields: a host identification and a key value identifying the DETAIL*IM instance that serves at the remote distributed set header for the distributed set at that host. The DETAIL*IM record contains two fields: a host identification of the owner record instance and a key value equivalent to the key of the owner instance. This key value serves as a CALC key for the record.

2.4 Summary

In this chapter, schema facilities to support the definition of a global schema and its associated mapping to local schema structures were defined along with a possible syntax. The use of a global schema mechanism provided geographic transparency for the application programmer. The proposed schema architecture consisted of global external schemas, a global conceptual schema, a global internal schema, and multiple local schemas (interface, conceptual, and internal). This multiple schema architecture extends the ANSI/SPARC three-schema architecture to the distributed environment. Using a top-down process the global schema processor generated the local schema DDL definitions from a given global internal DDL definition.

Local schema structures were classified as homogeneous or heterogeneous. With heterogeneous schema structures it was possible to have distributed sets, sets whose owner and member instances were not necessarily collocated. An implementation of distributed sets was proposed which formed the basis of a distributed directory. Either physical or symbolic pointers could be used to define distributed set linkages. An entry-record directory was proposed as an alternative to sequential scan or polling to find transaction entry-records.

Figure 2-13. Supplier D/B Global Internal Schema

```
GLOBAL INTERNAL SCHEMA SUPPLY HETERO PRIVACY LOCK IS "MFGVP"

DATABASE  PARTS
  HOSTS ARE BOSTON, CHICAGO, HOUSTON, SEATTLE.

DATABASE SUPPLY
  HOSTS CHICAGO.

AREA IS PARTSF COMPONENTS AT ALL  SIZE IS 40 PAGES.

AREA IS SUPPLF COMPONENTS AT CHICAGO  SIZE IS 15 PAGES.

RECORD PART  LOCATION MODE DYNAMIC-LOCAL
     KEYED USING PARTNO  DUPLICATES GLOBAL  WITHIN PARTSF.
PARTNO CHARACTER 5.
PARTNM CHARACTER 20.
INVQTY FIXED.

RECORD SUPPLIER  LOCATION MODE STATIC
     KEYED USING SUPPNO  WITHIN PARTSF.
SUPPNO CHARACTER 5.
SNAME CHARACTER 20.

RECORD DETAIL LOCATION MODE STATIC
     VIA SUPPLIES  WITHIN SUPPLF.
SLINE FIXED.
SPNO CHARACTER 5.
SQTY FIXED.

SET SUPPLIES
     MODE IS CHAIN
     OWNER SUPPLIER MEMBER DETAIL
     ORDER IS FIRST.

SET SUPPBY   DISTRIBUTED
     MODE IS CHAIN
     OWNER  PART   USING  PNO
     MEMBER DETAIL LINKED TO OWNER USING PNO
     ORDER IS FIRST.
```

Figure 2-14. Supplier Database Local Schemas

```
   -------------
   |  PART      |
   -------------
     |
     | SUPPBY*IO
     |
     V
   -------------
   |  PART*IO   |
   -------------
```

```
          ----------------       -------------------
          | DETAIL*IM     |      |    SUPPLIER       |
          ----------------       -------------------
                  |                    |
                  | SUPPBY*IM          |  SUPPLIES
                  |                    |
                  V                    V
                 ----------------------
                 |       DETAIL        |
                 ----------------------
```

Figure 2-15. Part D/B Local Internal Schema DDL

```
*HOSTID:BOSTON  (similiar for CHICAGO, HOUSTON, and SEATTLE)
*
SCHEMA PARTS  PRIVACY LOCK IS "MFGVP".

AREA IS  PARTSF  SIZE IS 10 PAGES.

RECORD PART  CALC USING PARTNO DUPLICATES NOT WITHIN PARTSF.
PARTNO CHARACTER 5.
PARTNM CHARACTER 20.

RECORD PART*IO LOCATION MODE VIA SUPPBY*IO.
PART*IH     TYPE FIXED.          !MEMBER HOST SITE.
PART*IK     TYPE CHARACTER 5.    !PARTNO

SET SUPPBY*IO
     MODE IS CHAIN
     OWNER PART  MEMBER PART*IO
     ORDER FIRST.
```

Figure 2-16. Supply D/B Local Internal Schema DDL

```
*HOSTID:CHICAGO
*
SCHEMA SUPPLY PRIVACY LOCK IS  "MFGVP".

AREA IS SUPPLF SIZE IS 15 PAGES.

RECORD SUPPLIER  CALC USING SUPPNO DUPLICATES NOT  WITHIN
SUPPLF.
SUPPNO TYPE CHARACTER 5.
SNAME TYPE CHARACTER 20.

RECORD DETAIL  VIA SUPPLIES  WITHIN SUPPLF.
SLINE TYPE FIXED.
SPNO TYPE CHARACTER 5.
SQTY TYPE FIXED.

RECORD DETAIL*IM  LOCATION MODE CALC USING DETAIL*IK.
DETAIL*IK       TYPE CHAR 5.    !RECORD KEY
DETAIL*IH       TYPE FIXED.     !HOST ID OF DIST. SET OWNER
SITE

RECORD DETAIL*IV  LOCATION MODE VIA SUPPBY*IV.
DETAIL*IH TYPE FIXED.  !OWNER HOST SITE


SET SUPPLIES
     MODE IS CHAIN
     OWNER SUPPLIER MEMBER DETAIL
     ORDER FIRST.

SET SUPPBY*IM
     MODE IS CHAIN
     OWNER DETAIL*IM
     MEMBER DETAIL LINKED TO OWNER
     ORDER FIRST.
```

3

Data Manipulation Facilities

3.0 Introduction

The data manipulation language defined for CODASYL Database Management Systems is relatively low-level and procedural. In order to ease the programmer's burden and to introduce the potential for optimization, particularly important in distributed database management, data languages with a high-level, non-procedural focus are used. Toward this end, a data language, SEEDFE, based on a FOR EACH construction, is proposed.

Several data languages have influenced the development of SEEDFE: HI-IQ [Gerritsen 75b]; UNEQ [Prenner 77]; DATA TYPE RELATION for PASCAL [Schmidt 77]; EXTENDED DATABASE ARCHITECTURE [Date 76]; Q, developed by Hayward, Sangal, and Buneman [Hayward 78]; and VIRTUAL INFORMATION OBJECTS (VIO'S) [Clemons 76]. Each of these languages include operations at a level higher than the record level.

Section 3.1 defines the syntax and semantics of SEEDFE. The nature of the mapping from SEEDFE to CODASYL-SEED DML for a non-distributed database is discussed in this section. Section 3.2 presents an integrated example based on an order-parts database. The use of SEEDFE for distributed transaction processing is discussed in chapter 4.

3.1 The SEEDFE Data Language

The SEEDFE Data Language is implemented as an extension of a Structured FORTRAN Pre-Processor [Friedman 77]. The database programmer can intermingle SEEDFE database constructions and Structured FORTRAN. The SEEDFE pre-processor, which has access to database schema information, translates the SEEDFE constructions to standard FORTRAN and associated CODASYL DML.

The SEED CODASYL DBMS [SEED 77], which was used for prototype development, includes the following DML operations supported via a FORTRAN call interface: database open (DBOPEN), database close (DBCLOS), find using calc-key (FINDC), find searching area (FINDAP), find using set membership (FINDPO), find using membership in sorted set (FINDV), and find using a set of values (FINDU). SEED also includes update primitives to store a record (STORE), to modify a record (MODIFY), and to delete a record (DELETE). The SEED system supports additional funtionality which was not used in the prototype development, e.g., the manual set operations, INSERT and REMOVE, and the multiple set membership find command, FIND. This find command retrieves that set of member records which are simultaneously in multiple sets, where each set instance is defined by a key value.

3.1.1 Initiation/Termination Of Database Processing

Since a database must be opened prior to database processing, the SEEDFE data language includes an INITIATE command to open a sub-schema with a given password and usage mode:

```
INITIATE sub-schema, password, mode.
```

The INITIATE statement triggers a call to the database open command. A call to the more general open statement, INVOKE, is not supported.

The database is automatically closed upon leaving the program. Normally, with a Structured FORTRAN program the STOP statement directly precedes the FORTRAN END statement. The pre-processor generates the database close call when this END statement is detected.

3.1.2 FOR EACH Processing

The basic construct in the SEEDFE data language is the FOR EACH construct. For each record instance, defined by underlying access paths, get each record instance and process the body of the FOR EACH loop. In other words, the FOR EACH construct is a loop construct where the loop range is a set of record instances.

The syntax of the FOR EACH, abbreviated FE, is given below:

```
FE record-name [OF set-name] (condition)
      o
      o
            < Body of FE loop >
      o
      o
ENDFE
```

The user supplies record-name, and optionally set-name and condition. The set-name defines a specific access path where multiple paths may exist. The condition is used to qualify record selection. The ENDFE terminates the body of the loop.

The actual processing implied by the FE statement is now defined. If no "OF clause" is present, the system checks if the given record-name has CALC location mode. If the record has CALC location mode and the condition is of the form: "calc-key-name .EQ. value," the system uses the SEED FINDC (find using calc key) to find a record instance. If CALC duplicates were allowed, the system would iterate through the duplicate record instances with the same key value.

If the record did not have CALC location mode and no set was specified, the system uses the SEED FINDAP (find record type in area) to find the relevant records. If the record is in a singular set but the programmer did not specify a set-name, the system uses the SEED FINDPO (find record using set membership).

With an "OF clause" the system uses the set specified to find the records. If the record is an owner of the specified set, the system uses the SEED FINDO (find owner) to find the relevant record. If the record is a member of the set named, the system uses the SEED FINDPO to find the relevant records. Current database position determines the relevant set occurrence. If the set is sorted in the FINDPO case and the condition is of the form: "sort-key-name .EQ. value", a SEED FINDV (find using sorted set membership) would be more efficient than the FINDPO because the set member instances do not need to be searched to the end of the set but only as far as implied by "value." Sort key duplicates are handled analogously to calc key duplicates.

In all cases after an appropriate FIND, a SEED GET is performed to move the record into the UWA. Aside from sort or calc key equality, which has already been addressed, the condition string is used as a filter to qualify record selection. Only records which satisfy the condition test are allowed to pass through the filter to be processed by the body of the FE loop.

3.1.2.1 Multiple key retrieval. In many cases, particularly when setting up databases for information retrieval purposes, retrieval based on a set of key-value relationships must be performed. Looking ahead to this situation, the DBA will usually include a number of indexing sets to assist in servicing these retrieval requests. A typical schema structure for indexing retrieval is illustrated in figure 3–1.

In figure 3–1, records R1, R2, . . . Rj define set instances of sets S1, S2, . . . Sj, respectively. Given a group of key-values, chosen to define instances of a subgroup of the j indexing sets, the group of base-records which is

Figure 3-1. Indexing Set Retrieval of Base-Records

```
            ------           ------                 ------
K1 ---->   | R1 |   K2 ---->  | R2 |   ...  Kj ---->  | Rj |
            ------           ------                 ------
                \                |                    /
                 \               |                   /
                  \              |                  /
                   \             |                 /
                    \            |                /
            S1       \       S2  |     ...    Sj /
                      \          |              /
                       \         |             /
                        \        |            /
                         V       V           V
                         ----------------------
                         |    Base-Record     |
                         ----------------------
```

simultaneously in all these sets is determined. This set of base-records will be referred to as the *simultaneous set*.

The FE construct could be extended to allow specification of the sets to be used in determining the simultaneous set. For example,

FE base-record OF S1(C1), S2(C2), . . . Sj(Cj) (base cond).

With this syntactic form a condition for each branch of the access tree (Ci) is included plus one for the base-record (base condition). The sets used to determine the simultaneous set are explicitly named.

In many cases there is enough information in the condition string to determine the relevant sets. Knowing the key names, a check can be made to determine if the key is a field in a record directly or indirectly owning the base-record. If there is ambiguity about which set to use along an access path, the ambiguous portion of the path must be made explicit by giving the set names.

A branch of the retrieval tree can involve more than one record-set pair. If intermediate sets are involved on the respective retrieval tree branches, additional qualification on these records may be included.

Instead of having a base condition and conditions for each branch of the retrieval tree, all the conditions can be placed together in a single condition string, requiring the parser to decompose the single condition string into the respective condition strings.

It is often desirable to select a set of base-records determined by a range of values for each of the keys in the key-value set. In this case each condition for a record on an arm of the retrieval tree is of the form:

(K1.GE.value11 .AND. K1.LE.value1h).

3.1.3 The General FOR EACH Construction

The revised syntax for the FE construction with a single condition string and implicit set indication is:

FE base-record [OF S1, S2, . . . , Si] (condition)

The current implementation limits the condition string format to a disjunctive normal form, e.g., (X .AND. Y . ... .) .OR. (A .AND. .NOT.B . . .) .OR. . . . . The restriction to disjunctive normal form is done solely to simplify prototype implementation and is not necessary in a production implementation.

However, if each key occurs only once in the condition in a simple conjunctive form:

(keyname1.relation1.value1 .AND. keyname2.relation2.value2 . . .),

the condition can be mapped directly to the SEED FINDU (find using key values) command. This command allows the user to give a set of key names and associated relations. The system uses the current values in the user work area for the given keys and relations and searches for records satisfying all the relations simultaneously. Instead of specifying a value in the search string, an asterisk can be used to indicate that the relevant value for the item is in the user work area.

Because the SEED FINDU command uses the UWA to store the comparison value, each key can only be used once. For this reason range checking cannot be implemented directly with the FINDU command. To allow easy implementation of range evaluations and the more general disjunctive normal form, a new SEED command, FINDUU, was implemented. FINDUU is an extension to FINDU which allows a key to occur more than once. Since additional space is required to hold additional data values for the same key, FINDUU uses an additional table. For symmetry reasons all relevant data values are kept in the table.

3.1.4 Derived Data

Data can be derived from data present in a retrieval record or from records already retrieved and still present in the UWA. Count, average, min, max, and total are typical functions which are applied to those records which pass the filter condition of an FE command. To request the calculation of derived data, a DERIVE construct is used. All DERIVE statements must immediately follow their corresponding FE statement.

3.1.4.1 DERIVE Syntax. The syntax of the DERIVE is analogous to a simple assignment statement:

DERIVE var = [expr | CNT(r) | AVG(i) | TOT(i) | MIN(i) | MAX(i)].

3.1.4.2 DERIVE Semantics. When a DERIVE using internal system functions is parsed, code is generated to initialize the associated variable. The count function tabulates the number of record occurrences which pass the filter test, while the other functions (average, min, max, and total) perform their respective function on the indicated item. After each record is retrieved and passes the filter condition, all DERIVE functions associated with the FE loop are updated. As part of ENDFE processing, any finish processing is performed, e.g., the division to complete the average calculation. If the DERIVE is a simple assignment statement without these system functions, an assignment statement is generated after record retrieval just prior to entering the loop body. Results of the DERIVE calculation are available inside the loop body at the same and lower levels; however, on an average calculation the referenced value will be the running sum because the average is not computed until loop end. Since FORTRAN is a language where the variables are always available, derived values can also be referenced after leaving an FE loop. In this case the value retrieved is the last value assigned to the variable.

3.1.5 Data Derived Using Find Owner Accesses

The current implementation of SEEDFE requires the explicit enumeration of ambiguous access paths. In addition, if after finding a base-record and processing down its hierarchy of subordinate members, some data in a record reachable using find owner accesses are desired, then additional FOR EACH statements must be used to define access to that owner record. Although these "to owner" accesses may not be ambiguous, the current implementation of DSEED requires that they be explicitly specified. Two alternative extensions to SEEDFE are possible to remove this restriction. (However, as Eric Clemons points out, the explicit enumeration of set access paths yields an added degree of data independence. With implicit enumeration schema changes, adding new paths can invalidate existing, operational programs.)

The first alternative is to perform a very detailed language analysis. Once data items within an FE loop are identified and determined to be database items, a check can be made to determine if they are in records reachable using find owner accesses from the current record. Kaplan describes an algorithm, defined by Buneman, to perform this task [Kaplan 79]. If ambiguous paths exist, then the system cannot derive the access path without programmer interaction.

The second alternative is to extend the derive statement. In this case the derive statement is used to implement a CODASYL virtual item, an item reachable using find owner accesses. The programmer simply lists the relevant sets as part of the derive statement.

DERIVE v-item USING set-1, set-2, . . . set-k

If additional items are required using the same access path, they can all be retrieved at once by listing them along with v-item.

3.1.6 Recursive Traversal

Recursively processable structures can exist in the data instances of CODASYL data structures. A classic example is provided by the assembly-component structure used to represent a bill-of-material. Usually this structure is represented by two records and two sets. Figure 3–2 illustrates a typical schema structure to represent a recursively processable structure such as a bill-of-material. To traverse such a bill-of-material structure recursively a recursive form of the FE is used.

Figure 3–2. Bill-of-Material Structure

```
            ---------------
            |  R1 = PART  |
            ---------------
               |         |
    CALL-OUT   |         |  WHERE-USED
       S1      |         |      S2
               V         V
            ---------------
            |   R2=COMP   |
            ---------------
```

3.1.6.1 Recursive FE syntax

```
FE R1-name RECURSIVELY
                  DOWN S1-name TO R2-name UP S2-name
     [DEPTH d-variable] [LEVEL 1-variable]
     [ (condition) ]
   o
   o
   o
ENDFE
```

3.1.6.2 Recursive FE semantics. Starting with the current R1 instance, traverse down S1 to R2. Execute the body of the loop. During ENDFE processing, traverse up S2 and get another R1 instance. Continue this process until the loop has been traversed d-variable times (or as many levels that exist less than d-variable). Store the current level in the 1-variable. Use the optional condition filter to exclude an R1-R2 instance pair on a given loop pass. An internal stack and associated PUSH/POP functions support the implementation of this depth-first tree traversal.

3.1.7 Database Update

In order to support database update, the FE construction is augmented using the keywords STORE, MODIFY, or DELETE. If one of these keywords ends the FE statement, an internal flag is set for that FE loop and the associated DML statement is generated. For MODIFY, the SEED DML MODIFY statement is generated at the end of the FE loop; whereas, for STORE and DELETE, the appropriate SEED DML statement is generated prior to the entry of any inner FE loops. In all cases, the relevant class of records to be stored, modified, or deleted is identified by an associated FE loop.

3.1.7.1 STORE Keyword. If the STORE keyword ends an FE statement, the FE loop stores one data record instance on each loop iteration. The loop body contains FORTRAN statements to establish values for the relevant UWA items for the record to be stored. If a series of records of the same type are to be stored, the FE loop is embedded in a standard Structured FORTRAN WHILE (C) . . . ENDWHILE loop or UNTIL(C) . . . ENDUNTIL loop. Alternatively, the WHILE or UNTIL clause can be included in the FE statement.

If the condition of an FE store involves a key whose value will be defined in the body of the loop, an asterisk can be used in the value portion of the condition to indicate that the desired key value will be in the UWA. In some cases a key value required in the condition string will not be known until statements in the loop body are executed, e.g., a CALC-key value. In this case, an asterisk is placed in the condition key value field to indicate that the desired key will be placed in the UWA during the loop body.

Because of the procedural nature of manual set membership update operations, only automatic set membership is supported, i.e., the CODASYL statements, insert and remove, are not supported. For the same reason, the CODASYL ability to retrieve and insert member records positionally is not supported. All these features could be supported in SEEDFE by defining additional keywords to augment the FE structure.

3.1.7.2 MODIFY keyword. If the MODIFY keyword ends an FE statement, the data record is retrieved prior to entering the loop body and a call to the SEED DML MODIFY is generated at the end of the loop body. Placing statements changing selected item values in the loop body will cause those items to be modified in the database.

3.1.7.3 DELETE keyword. If the delete keyword ends an FE statement, the data record is found and moved to the UWA prior to entering the loop body. A call to the SEED DML DELETE is generated prior to entering any inner FE loops. A more detailed language analysis would indicate if the data record move to the UWA is necessary.

3.2 Integrated Example

An integrated example showing the use of SEEDFE using an order-parts database follows.

3.2.1 Order-Parts Database Schema Structure

Figure 3–3 contains the schema diagram for the order-parts database. Information on a given order is stored in the ORDERB record. Order line information is stored in the LINITM record. The LINES set represents the one-to-many relationship between an order record and its lines. Information on a given part is stored in the PART record. The PARTS set represents the one-to-many relationship between a part and the order lines referencing the part. The COMP and USEDON sets together with the QTY record represent the assembly-component, bill-of-material structure. Set COMP represents the direct components of a part; whereas, set USEDON represents (through owner references) the parts this part is used on. The QTY record holds

Figure 3–3. Order-Parts Database Schema Diagram

```
          --------------
          |   SYSTEM    |
          --------------
                 |
                 | ORDERS
                 |
                 V
          -------------------------      -----------------
Orderno->| ORDERB: orderno,custno| Partno->| PART: partno   |
          -------------------------      -----------------
                  |                    /      |    |
                  |                   /       |    |
             via  | LINES      PARTS /        |    |
                  |                 /         |    |
                  |                /    COMP  |    | USEDON
                  V               V           V    V
          -------------------------------      -------------
          |  LINITM: oprice,oqty,lineno |      | QTY: qty   |
          -------------------------------      -------------
```

information about the direct component relationship between two part instances, e.g., the number of units of each direct component.

3.2.2 Sample Transactions

Programs will be developed for four transactions processed uder a sub-schema, ORDSUB, derived from the given schema.

```
C      "Print the Total Order Backlog Dollar Value"
C
       INITIATE  ORDSUB, SSPASS, RETRIEVAL
       FE  ORDERB  OF ORDERS
            FE  LINITM  OF  LINES
                 DERIVE  VALUE  =  OPRICE * OQTY
                 DERIVE  OVALUE =  SUM ( VALUE )
            ENDFE
            DERIVE  BACKLG  = SUM ( OVALUE )
       ENDFE
       PRINT,BACKLG
       STOP
```

In this program the total value of each order is calculated by determining the sum of the value of an order line. This value is the quantity ordered times price. In turn, all orders are processed and a running total of the order backlog is maintained in BACKLG.

```
C    "Print List of Orders Affected by Part Shortage"
C
    INITIATE  ORDSUB, SSPASS, RETRIEVAL
    READ,POUTOF
    FE  PART  (PARTNO  .EQ.  POUTOF)
         FE  LINITM  OF  PARTS
            FE  ORDERB  OF  LINES
                           PRINT,ORDRNO,CUSTNO,LINENO
            ENDFE
         ENDFE
    ENDFE
    STOP
```

In this program the part record identified by POUTOF is retrieved. All records in the PARTS set and associated owner instances in the ORDERS set are then retrieved.

```
C  "Print a full indented Bill-of-material explosion"
C
  INTEGER SPACE (20)
  DATA SPACE / 20 * '     ' /
  INITIATE  ORDSUB,  SSPASS, RETRIEVAL
  READ,PARTE
  FE  PART  (PARTNO .EQ. PARTE)
        FE PART RECURSIVELY DOWN CALL-OUT TO COMP
                    UP WHERE-USED LEVEL L
           PRINT,L,(SPACE(I),I=1,L),PARTNO,QTY
        ENDFE
  ENDFE
  STOP
```

A full explosion of PARTS is generated in this program. The returned level variable, L, is used to indent the print-out.

```
C      "Add an Order"
C
     INITIATE ORDSUB, SSPASS, UPDATE
     FE ORDERB (ORDERNO .EQ. *) STORE
          READ, ORDERNO, CUSTNO
          FE LINITM STORE WHILE (LINENO.GT.0) DO
               READ, LINENO, OPRICE, OQTY
          ENDFE
     ENDFE
     STOP
```

In this program an order is added to the database. The inner FE loop enters a LINITM record until a value of negative one for LINENO is read.

3.3 Summary

In this chapter the syntax and semantics of SEEDFE, a high-level FOR EACH data language for CODASYL, were described. SEEDFE was implemented using a modified Structured FORTRAN Pre-Processor. SEEDFE as described in this chapter was implemented in a non-distributed environment. In the next chapter a less procedural view of the language is taken and the language is extended to a distributed environment.

SEEDFE included a FOR EACH construction to identify members of a set (CALC-set, AREA-set, or CODASYL-set); a DERIVE statement to identify data derived using system functions (count, average min, max, sum) or user functions; and update facilities. The FOR EACH construction also was extended for multiple key retrieval and recursive traversal.

4

DSEED Distributed Transaction Decomposition

4.0 Introduction

Portions of the prototype described in this study have been implemented on the Wharton School DEC KL10 under the TOPS-10 (6.03) operating system with data management services provided by the SEED DBMS [SEED 77]. In this chapter the relationship between the high-level data language, the global schema mechanism, and the implementation environment is described. The methodology by which a transaction, represented by a FOR EACH program, is decomposed into front-end and back-end tasks by a preprocessor is the main focus of the chapter. Since this description is dependent on the implementation environment, the overall architecture of the DSEED prototype is summarized in section 1. In section 2, the distributed task decomposition process is defined and a sample order-parts transaction introduced. In section 3, the results of decomposition for the sample transaction are described. In section 4, the use of an external schema to represent transaction programming is demonstrated. In section 5, improvements to the DSEED stream processing mechanisms are proposed.

4.1 DSEED Prototype Architecture Overview

The DSEED prototype architecture is an example of a top-down distributed DBMS architecture. The architecture includes front-end control logic and back-end database service logic. This structure is summarized in figure 4–1. Each user transaction is decomposed into a front-end control task and one or more back-end support tasks. The application programmer is not concerned with this decomposition.

The front-end process, illustrated at the top of the figure, has four major components: a control point module for the front-end process at the host;

Figure 4–1. DSEED Prototype System Overview

```
                      ---------------
                      | HOST FRONT-  |
                      | END CONTROL  |
                      |  PROCESS     |
                      ---------------
                      | CONTROLLING  | - - - -(GLOBAL SCHEMA)
                      |     TASK     |
                      ---------------
                      | DDBMS FRONT_ |
                      |  END PROC'S  |
                      ---------------
                      | NETWORK-NCP  |
                      ---------------
                             |
                             |
                             |
               -------------------------------
               |      MESSAGE SERVICE         |
               -------------------------------
                     |                   |
                     |                   |
                     |                   |
         ---------------          ----------------
  D/B -  | DML   | NCP |   D/B -  | DML    | NCP  |
         ---------------          ----------------
 (LOCAL  | DDBMS BACK- | (LOCAL   | DDBMS BACK-   |
 SCHEMA) |  END PROC'S | SCHEMA)  |  END PROC'S   |
         ---------------          ----------------
         | CONTROLLED  |          | CONTROLLED    |
         |   TASK      |          |    TASK       | o o o
         ---------------          ----------------
         | HOST BACK-  |          | HOST BACK-    |
         |  END CONTROL|          |  END CONTROL  |
         |   PROCESS   |          |    PROCESS    |
         ---------------          ----------------
```

the controlling task module, which is the front end control task of the user's transaction; DDBMS front-end procedures, which support the operation of the controlling task; and the network control program interface, NCP.

The back-end process, illustrated at the bottom of the figure, has five major components: a control point module for the back-end process at the host; the controlled task module, which is the collection of back-end tasks to support the user's transaction at that host; the DDBMS back-end procedures, which support the operation of the controlled task; the SEED DBMS DML; and the network control program interface, NCP.

A compiled form of the GLOBAL schema is used by the controlling task to guide the operation of the internal interface, the software which creates the global view from the local views. Each controlled task component uses its associated local schema to control its actions.

The controlling task communicates with the back-end tasks using a process-to-process protocol described later in this chapter. This data transfer protocol, the *DDBMS Protocol*, is built using the DEC10 InterProcess Communications Facility, which provides a message sending/receiving service between DEC10 processes. Routines, developed at Rutgers University using the InterProcess Communication Facility, were augmented to provide a simplified, machine-independent network interface:

o NCPOPN—Open a connection to a process
o NCPSND—Send a message to a process
o NCPRCV—Receive a message
o NCPCLS—Close a connection to a process

A multiple machine environment was simulated on the single DEC10 host. A host in this environment is simulated by a number of jobs. Each simulated host in the network had a front-end and a back-end control point with known names. The simulated network is initialized by starting these control points, i.e., by logging in DEC10 jobs for each front-end and back-end control point. The task image of a defined user transaction is initiated by giving the front-end control point the name of that transactions's task image file. The front-end control point starts that program, which in turn starts relevant back-end supporting tasks. This process is described in more detail later in this chapter.

The process of initializing the network is facilitated by the use of a DEC10 utility, OPSER. This utility allows a master job to start subordinate jobs by command. The start-up procedure can be activated from a file of commands, simplifying the start-up process. All terminal input/output from the master and subordinate jobs is time-stamped and placed in a disk log file. The log file facility proved to be very useful in debugging communications protocols.

To support process-to-process communications a unique process ID is required. A process ID was constructed by the string concatenation of global host name, global external schema name, local host name, local job number, and channel priority (R for regular, E for express). Currently, only a regular priority channel is used.

4.2 Distributed Task Decomposition

4.2.1 The FOR EACH User Transaction

A user's transaction is represented by a SEEDFE program. This SEEDFE program is decomposed by a pre-processor into front-end and back-end tasks, which are then installed on the respective network hosts. The pre-processor implementation and associated pre-installation of tasks was done to facilitate prototype implementation and to improve performance. Execution-time compilation of the transactions could have been performed, although pre-installation of large, known, frequently-run transactions is certainly more efficient. This task decomposition process is summarized in figure 4–2.

Figure 4–2. SEEDFE Pre-Processor Task Decomposition

```
SEEDFE program                    GLOBAL SCHEMA
               \                 /
                \               /
                 \             /
                  V           V
        -------------------------
        |  SEEDFE Pre-Processor  |------> Listing
        -------------------------
                  /            \
                 /              \
                /                \
               V                  V
        Front-end           Back-end
         Control             Controlled
          Task                Task(s)
```

The SEEDFE pre-processor is guided by the global schema. Local schema information is not necessary because of the close structural relationship of the global and local schemas. Local schema information is only required by the back-end processors.

4.2.2 SEEDFE USING/RETURNING Clauses

The SEEDFE FOR EACH language syntax, introduced in chapter 3, has been extended slightly with the introduction of :USING and :RETURNING clauses. These clauses describe what data is used or returned by the associated FE loop. The use of these clauses highlights the data flow requirements of a transaction and simplifies a pre-processor implementation. A production implementation could determine this information directly using a more detailed syntactic analysis and the variable scope determination of block structured languages such as ALGOL, PASCAL, and PL/I.

4.2.3 Distributed FE Task Example

To illustrate the interrelationship of the various DSEED DDBMS components, a common example will be used throughout this chapter. A global schema, based on a portion of the schema illustrated in chapter 3, was used. The global schema DDL for this example (the ORDERB-LINITM-PART confluency) is included in figure 4–3. The local schema definitions to support this structure are included in figure 4–4.

The distributed DDL parser for the global schema DDL described in chapter 2 was not implemented. For testing purposes the distributed DDL

Figure 4–3. Order-Parts Global Schema DDL Definition

```
*ORDRDG.DDL
*DDBMS GLOBAL SCHEMA DEFINITION.
*HOSTID: PHILA
GLOBAL INTERNAL SCHEMA  ORDRDG  HETERO
  PRIVACY LOCK IS DBA.

HOSTS ARE PHILA BOSTON CHICAGO.

AREA IS ORDERF COMPONENTS AT BOSTON.
AREA IS PARTF  COMPONENTS AT CHICAGO.

RECORD  ORDERB  LOCATION MODE STATIC  WITHIN ORDERF
                KEYED USING ORDERNO DUPLICATES NOT.
  ORDRNO  TYPE  CHARACTER 5.
  CUSTNO  TYPE CHARACTER 5.
  CUSTNM  TYPE  CHARACTER 20.

RECORD  LINITM LOCATION MODE STATIC  WITHIN ORDERF
               VIA LINES.
  LINENO  TYPE  FIXED.
  OQTY    TYPE  FIXED.
  OPN     TYPE  CHARACTER 5.
  OPRICE  TYPE  REAL.

RECORD  PART  LOCATION MODE STATIC  WITHIN PARTF
              KEYED USING PARTNO DUPLICATES NOT.
  PARTNO  TYPE  CHARACTER 5.
  PQTY   TYPE  FIXED.
  PDESC   TYPE  CHARACTER 30.

SET LINES
  MODE IS CHAIN
  ORDER SORTED
  OWNER ORDERB MEMBER LINITM  MANDATORY AUTOMATIC
  ASCENDING KEY LINENO
  SET SELECTION IS THRU LOCATION MODE OF OWNER.

SET PARTS   DISTRIBUTED
  MODE IS CHAIN
  ORDER IS FIRST
  OWNER PART  MEMBER LINITM  MANDATORY AUTOMATIC
  SET SELECTION IS THRU LOCATION MODE OF OWNER.
```

Figure 4-4. Order-Parts Local Schema DDL Definitions

```
*DDBMS LOCAL SCHEMA.
*HOSTID: BOSTON
SCHEMA ORDRDL  PRIVACY LOCK IS DBA.

RECORD ORDERB LOCATION MODE CALC USING ORDRNO DUPLICATES NOT
  ORDRNO   TYPE  CHARACTER 5.
  CUSTNO   TYPE  CHARACTER 5.
  CUSTNM   TYPE  CHARACTER 20.

RECORD LINITM  LOCATION MODE VIA LINES.
  LINENO  TYPE  FIXED.
  OQTY    TYPE  FIXED.
  OPN     TYPE  CHARACTER 5.
  OPRICE  TYPE  REAL.

RECORD PARRIM LOCATION MODE CALC USING PARIIM DUPLICATES NOT
  PARIIM  TYPE  FIXED.

SET LINES
  MODE IS CHAIN
  ORDER SORTED
  OWNER ORDERB MEMBER LINITM  MANDATORY AUTOMATIC
  ASCENDING KEY LINENO
  SET SELECTION IS THRU LOCATION MODE OF OWNER.

SET PARSIM  MODE IS CHAIN  ORDER FIRST
  OWNER PARRIM MEMBER LINITM
  SET SELECTION IS THRU LOCATION MODE OF OWNER.
------------------------------------------------------------

*DDBMS LOCAL SCHEMA.
*HOSTID: CHICAGO
SCHEMA PARTDL  PRIVACY LOCK IS DBA.

RECORD PART  LOCATION MODE CALC USING PARTNO DUPLICATES NOT
  PARTNO  TYPE  CHARACTER 5.
  PQTY   TYPE  FIXED.
  PDESC   TYPE  CHARACTER 30.

RECORD PARRIO LOCATION MODE VIA PARSIO.
*          ID OF HOST WITH DISTRIBUTED SET COMPONENTS
  PARIIH  TYPE  FIXED.
*          DATABASE KEY OF LOCAL SET HEADER
  PARIIO  TYPE  FIXED.
SET  PARSIO  MODE IS CHAIN   ORDER FIRST
     OWNER PART   MEMBER  PARRIO
     SET SELECTION IS THRU  LOCATION MODE OF OWNER.
```

definition for this example, without the special clauses, was compiled using the standard SEED schema processor utility, program FDP. A special load program, specific to this example, took the results of this compilation and added the information which would have been derived from the global DDL phrases, from FORTRAN DATA statements, and produced a file containing the compiled form of the schema. The changes to the schema parser to create the required composite file directly were investigated and found to be a straightforward, but time-consuming programming task.

4.2.4 Sample Order-Parts Transaction

Consider the following transaction: "Print the orders including part 12345, where the quantity ordered exceeds 100 units." The associated FE program for this transaction is included in figure 4–5.

Figure 4–5. Distributed Order-Parts Transaction

```
FE PART (PARTNO .EQ. "12345")       :using partno
    FE LINITM OF PARTS   (OQTY .GE. 100)
        FE ORDERB OF LINES
          WRITE(5,100)PARTNO,PDESC,LINENO,OPRICE,ORDERNO
100                (1X,A5,1X,6A5,1X,I5,1X,F8.2,1X,A5)
        ENDFE      :returning orderno
    ENDFE      :returning lineno,oprice
ENDFE      :returning pdesc
```

4.2.5 Data Flow Graphs

Every distributed transaction has an associated data flow graph, which illustrates the sources and uses of data items. The data flow graph of the transaction just described can be found in figure 4–6. In this diagram the boxes are tasks and the arrows are the flow of data. In this example the controlling task is in Philadelphia; the back-end tasks are in Boston and Chicago. The "U" and "R" indicate used and returned data respectively.

Depending upon how the *control point*, the locus of control code for a task, is implemented, different control structures are possible. Two kinds of control structures are possible with this data flow structure. With a *static control point*, the controlling task is always in direct control of all back-end tasks. All data flow occurs between the front-end control task and back-end tasks; no data is directly sent between back-end tasks. With a *dynamic control point*, control may be temporarily passed to a subordinate back-end task which serves as an agent of the front-end task to supervise other back-end tasks. In this case, data can flow directly between back-end tasks. The use of dynamic control points can lead to improved performance because of a reduction in the number of messages or the cost of sending those messages by sending larger, composite messages. However, the dynamic control point software is more complicated and recovery after system failure is more difficult.

Figure 4-6. Sample Data Flow Graph

```
          ----------          ----------          ----------
 ------->|          |--------->|          |--------->|          |
U:PARTNO |          | U:none   |          | U:none   |          |
         |          |          |          |          |          |
         |  PART    |          |  LINITM  |          |  ORDERB  |
         |          |          |          |          |          |
R:PDESC  |          | R:LINENO,|          | R:ORDER# |          |
 <-------|          |<---------|          |<---------|          |
         |          |  OPRICE  |          |          |          |
          ----------          ----------          ----------

 User.    Task 1.             Task 2a.            Task 2b.
 Phila.   Chicago             Boston              Boston
```

4.2.6 DDBMS Protocol

The correct data transfer implied by the data flow graph is controlled by using a process-to-process protocol [Crocker 72]. An overview of the DSEED DDBMS protocol can be found in figure 4-7. This protocol is embedded in the front-end/back-end tasks generated by the SEEDFE pre-processor. A front-end task must initiate a transaction, invoke one or more subordinate tasks, and finish a transaction. Each of these actions has companion actions on one or more back-end machines. Execution of a subordinate task involves the transfer of data. This data is transferred using a data stream mechanism.

Figure 4–7. DDBMS Protocol Overview

```
  FRONT-END                   BACK-END,
                              one illustrated


------------                ------------
| INITIATE |<--control--->|  DBOPEN   |
------------                ------------


------------                ------------
|  INVOKE  |<---control-->| back-end  |
|   TASK   |<--DATA XFER->|user tasks|
------------                ------------



------------                ------------
|  FINISH  |<--control--->|  DBCLOS   |
------------                ------------
```

4.2.6.1 FE task decomposition logic. An FE program is decomposed into subordinate tasks, one for each FOR EACH loop. The outer FE loop determines how the transaction is entered. Once a transaction is entered, record access is determined by the underlying set access paths.

Relevant hosts along the access path are determined at execution time. Figure 4–8 summarizes the decision logic possibilities to establish the relevant hosts.

The logic summarized in this figure is embedded in a subroutine called at execution time. The subroutine leaves the relevant host information in a data table which all DDBMS procedures access as required. When owner or member record instances of distributed sets are accessed, the distributed directory information is returned with the record data.

4.2.6.2 Data stream format. A *data stream* is used to transfer user data. The using/returning data flow indicated in figure 4–6 and the data transfer between the "invoke task" and "back-end user tasks" indicated in Figure 4–7 are implemented with a data stream mechanism. In the simplest case, each FE loop of a transaction generates one data stream. A data stream can be derived from several back-end supporting tasks. In this case, the supporting streams are combined by the internal interface so that only a single stream is seen on the global side of the internal interface.

A data stream is identified by a header, a body, and a tail. The header includes stream body-format information. The tail is the end-of-stream marker. The tail can also include information derived from the stream as it was produced, e.g., derived data such as the sum of a data item. The stream body can include data derived from an individual record fragment, called a *chunk*, a group of chunks, or all relevant chunks for the stream. The current implementation of DSEED supports only single chunks. The use of whole stream bodies will be discussed later in this chapter.

Figure 4–8. FE Task Host Determination Logic Tree

```
Entry-Point

   Static -- Use fixed location

   Dynamic

      If no directory -- Poll Possible Locations
      If directory -- Search directory

Set Access Along Path

   Distributed Set

      To Owner -- Use host pointer in im record
      To Member -- Use host pointers in io records

   Non-Distributed Set

      To Owner -- Use location of member instance
      To Member -- Use location of owner instance
```

4.2.7 Front-End/Back-End Task Structure

The SEEDFE pre-processor generates FORTRAN programs for the front-end and back-end tasks. The front-end control task includes an initiation section, a program body, and an end section. The initiation section sets up the task control tables by calling subroutine DDBSTT. This subroutine is generated by the SEEDFE pre-processor after processing the FE program. The program body is a series of embedded loops, one for each original FE loop, with the exception that records which can be reached by doing local find owner accesses do not require an additional task. In our example, access to the ORDERB record from the LINITM record does not yield an embedded loop. The end section closes the database.

The back-end tasks parallel the front-end tasks in structure. Each back-end task has a control portion which establishes communications with the front-end, opens and closes the database, and invokes tasks. The code for the tasks is found in a sub-routine DDBBUT, the back-end user task module. This module is generated by the SEEDFE pre-processor as the FE transaction is compiled. Both front-end and back-end program bodies use support routines which implement the DDBMS protocol.

4.2.7.1 DDBMS front-end support procedures. A front-end task body is supported by the following procedures: front-end-find-using-key (FNDUKF), front-end-find-using-set (FNDUSF), and get-data-chunk (GETCNK). FNDUKF and FNDUSF initiate a data record retrieval using a key value or record type match and using a set, respectively. GETCNK gets the relevant data items from the stream buffer and moves them to the UWA. The logical variable, end-of-stream, EOSTRM (level), is set .TRUE. when the associated stream of data is exhausted.

4.2.7.2 DDBMS back-end support procedures. A back-end task body is supported by the following procedures: back-end-find-using-key (FNDUKB), back-end-find-using-set (FNDUSB), build-chunk (BLDCNK), send-chunk (SNDCNK), send-tail (SNDTAI), and the back-end DBMS DML. FNDUKB and FNDUSB find and get a record instance using a key (not a set) and using a set, respectively. FNDUKB issues a DML call to find-using-calc-key (SEED FINDC) or find-record-in-area (SEED FINDAP), as required. FNDUSB issues a DML call to find-using-set-membership (SEED FINDPO), find-using-set-ownership (SEED FINDO), or find-using-a-sorted-set (SEED FINDV), as required. In all cases, to move a found record to the internal UWA, a get-data (SEED GET) is used. BLDCNK packs data using the stream format into a data message. SNDCNK sends the data message. SNDTAI packs and sends end-of-stream information (derived data such as the sum of a data item) and the end-of-stream marker.

Currently, each stream chunk is sent in its own message and acknowledged. This is convenient for control purposes and ease of implementation, but counterproductive to reducing network traffic. Improvements to this procedure are considered later in this chapter.

4.2.8 DDBMS Protocol Details

Further detail regarding the operation of the DDBMS protocol is included in figure 4-9. This figure expands the central portion of figure 4-7, which indicated the context of the data transfer process. Figure 4-9 outlines the data transfer protocol between the front-end control and associated back-end support (controlled) tasks for a single FE loop.

The control host determines which back-ends must be involved. This occurs by subroutine call at execution-time. The stream format is established (STARTT) and all involved hosts are polled for readiness (STRTPT). Establishing the format of the stream prior to stream transmission reduces the overhead of the individual stream chunks.

Once all back-ends have acknowledged acceptance, they are all instructed to start sending data by a request to "get-the-stream-header" (GETSHD). The back-ends build up the stream bodies, as required, and send them to the front-end using the "send-chunk" procedure (SNDCNK). At the front-end stream data for one record instance is retrieved and moved into the user work area by the "get-chunk" procedure (GETCNK). Each back-end stream is closed off by sending a tail identifier (SNDTAI).

Each message sent includes a message type. This type field allows the front-end and back-end processes to remain synchronized. In Figure 4-9, the following type codes are illustrated: INVK, invoke, i.e., place in ready status, a specific back-end task; INVH, start a ready task sending data; INVC, send a stream chunk; and INCT, send a stream tail. In addition, a SUBP code was used to instruct a back-end control point to start-up a back-end process; and, a SDBP code was used to instruct a back-end process to shut-down.

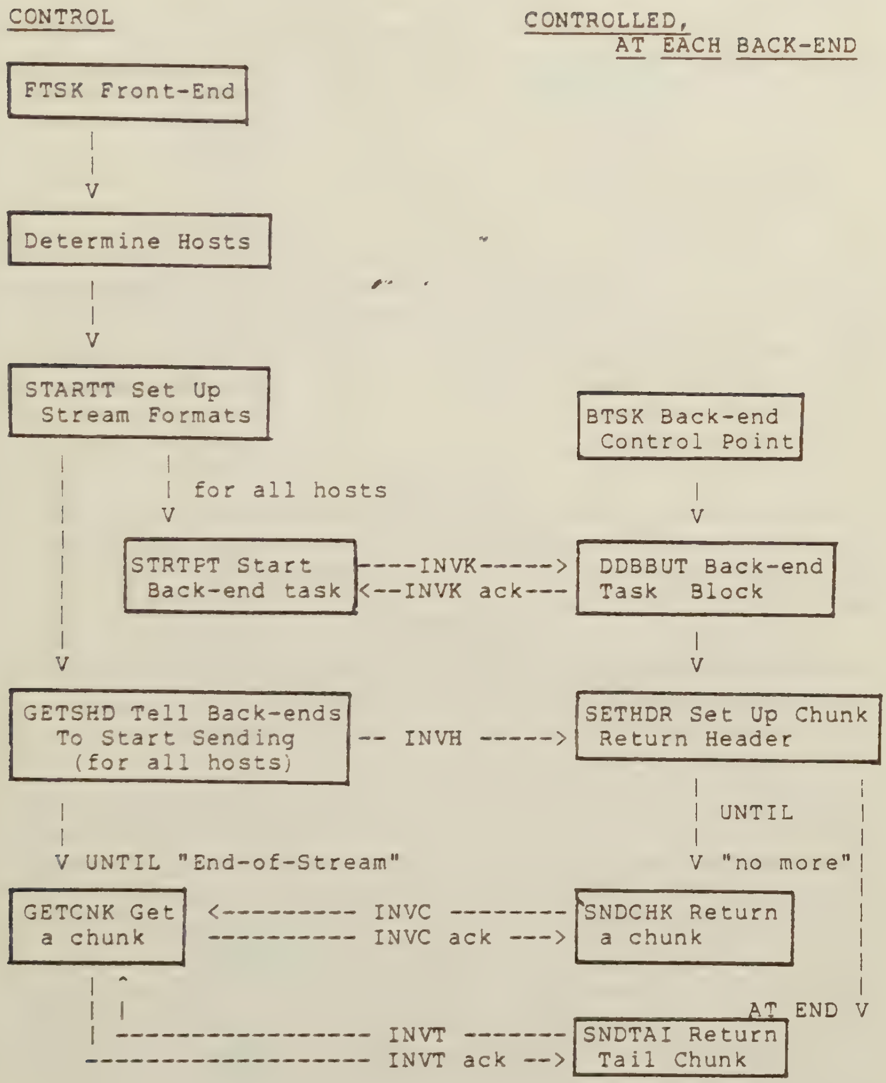
4.2.9 SEEDFE Pre-Processor Code Generation

The code generated by the SEEDFE pre-processor in a distributed task mode has special distributed FE loop entry and exit code routines. The actual implementation of changes to SEEDFE was only carried as far as was necessary to understand how it would actually be accomplished. In addition, it is simpler to generate all back-end tasks to include the same set of operations, but to only invoke the relevant ones. However, in the tasks presented below for our example, the irrelevant operations have been removed to clarify the presentation.

4.2.10 SEEDFE/Task Common Blocks

Several FORTRAN common blocks are used by the SEEDFE pre-processor and generated front-end and back-end tasks. The DDBSCH.COM common

Figure 4-9. DDBMS Protocol Details


CONTROL
CONTROLLED, AT EACH BACK-END
FTSK Front-End
Determine Hosts
STARTT Set Up Stream Formats
for all hosts
BTSK Back-end Control Point
STRTPT Start Back-end task
INVK
INVK ack
DDBBUT Back-end Task Block
GETSHD Tell Back-ends To Start Sending (for all hosts)
INVH
SETHDR Set Up Chunk Return Header
UNTIL "End-of-Stream"
UNTIL "no more"
GETCNK Get a chunk
INVC
INVC ack
SNDCHK Return a chunk
AT END
INVT
INVT ack
SNDTAI Return Tail Chunk

NOTE: ack indicates acknowledgement.

block includes the distributed schema structure, represented as the additional information in a global schema and the standard internal SEED common block. The DDBFCT.COM common block contains tables to control outstanding tasks, message work areas, and miscellaneous information. Files with a .WRK extension are database UWA definition statments. The DDBBCT.COM common block contains using/returning list information, message work areas, and miscellaneous information.

4.3 Sample Decomposition for Order-Parts Transaction

Figures 4–10, 4–11, and 4–12 include the front-end and back-end tasks which have been generated by the SEEDFE pre-processor for our sample transaction of figure 4–5. In these programs ?ERROR is used to indicate a place where the existence of an error condition should be checked. These error branches were not shown to improve the readability of the program.

4.3.1 FTSK10 Program Description

Program FTSK10 is the front-end control program for task 1. It consists of a main program and a sub-routine DDBSTT to set-up the task control tables. The main program has an initiation section, a program body, and a close section.

The initiation section invokes the DDBSTT routine to set-up the task control tables for the program and invokes the DDBFIN routine to instruct the relevant back-end hosts to get ready. At this point, if a required host is not available, suitable error logic would implement the desired recovery procedures.

The program body is a series of loops, generally one for each FE of the original program. Since the required ORDERB information can be reached from a collocated owner record, LINITM, the inner two loops of the original transaction are consolidated into a single loop, FE level, of the front-end control task. Each of these FE tasks will now be described.

In the FE level 1 task, the PARTNO field in the UWA is initialized to the desired "12345." Since access to a keyed record is required, a call is made to FNDUKF passing parameters indicating global record PART using global item PARTNO. The 10000 loop is the actual FE level 1 loop. Within this loop a call to GETCNK returns a chunk from stream 1 and sets the current status of the stream in EOSTRM(1). When the stream is exhausted, the loop test fails and control passes to the close section. There is no user body for FE task 1, beyond the embedded FE loop at level two.

In the FE level 2 loop, since access to a record by set is required, a call is made to FNDUSF passing a parameter indicating global set PARTS. In the

Figure 4–10a. Sample Front-End Task

```
      PROGRAM FTSK10
      IMPLICIT INTEGER (A-Z)
C          CONTROLLING TASK UNDER  ORDRDG AT HOST G
      INCLUDE 'DDBSCH.COM'     !GLOBAL SCHEMA INFO
      INCLUDE 'DDBFCT.COM'     !FRONT-END TASK CONTROL
      INCLUDE 'ORSSDG.WRK'     !D/B UWA DEFINITION
      LOGICAL EOSTRM(2)
C
C     INITIATE.
      CALL DDBSTT
      CALL DDBFIN
C      ?ERROR---GOTO 99998
C
C     --- PROGRAM BODY ---
C
C     - FE LEVEL 1 -
C
      PARTNO="12345"
      CALL FNDUKF(PART,PARTNO)        !PART RECORD
10000 CALL GETCNK(1,EOSTRM(1))
C        USER BODY OF FE LEVEL 1
C
C     - FE LEVEL 2 -
C
      CALL FNDUSF(PARTS)            !LINITM RECORD
10001 CALL GETCNK(2,EOSTRM(2))
C        USER BODY OF FE LEVEL 2
C
      WRITE(5,100)PARTNO,PDESC,LINENO,OPRICE,ORDERNO
100    FORMAT(1X,A5,1X,6A5,1X,I5,1X,F8.2,1X,A5)
C
      IF(.NOT.EOSTRM(2))GO TO  10001  !ENDFE 2
      IF(.NOT.EOSTRM(1))GO TO  10000  !ENDFE 1
C
C
C     --- END OF MAIN PROGRAM BODY ---
C
C     CLOSE.
99999 CALL DDBFCB
C      ?ERROR---GOTO 99998
      STOP
C
99998      PRINT*,'ERROR:',DDBSTA
           GO TO 99999
      END
```

Figure 4-10b. Sample Front-End Task

```
      SUBROUTINE DDBSTT
C       SET UP TASK CONTROL TABLES.
      INCLUDE 'DDBFCT.COM'     !FRONT-END TASK CONTROL
      INCLUDE 'DDBSCH.COM'     !GLOBAL SCHEMA INFO
      CALL DDBCTC  !CLEAR CONTROL TABLES.
C
      TASKNM='TASK1    '
      SCHNAM='ORSSDG    '
      PWORD='DBA   '
C
      ULIST(1,1)=8    !PARTNO
      RLIST(1,1)=10   !PDESC
      RLIST(1,2)=4    !LINENO
      RLIST(2,2)=7    !OPRICE
      RLIST(1,3)=1    !ORDRNO
C
      RETURN
      END
```

Figure 4-11a. Sample Back-End Task Control at Boston

```
      PROGRAM BTSK11
      IMPLICIT INTEGER (A-Z)
C     BACKEND TASK 1 AT HOST 1, UNDER ORDRDG.
      INCLUDE 'DDBBCT.COM'     !BACK-END TASK CONTROL
      GHSTID='PHILA    '
      GSNAME='ORSSDG    '
      LHSTID='BOSTON    '
C
      CALL DDBBSU(STA)   !ESTABLISH RETURN COMMUNICATIONS
      IF(STA.EQ.0) GO TO 90000
C       ?ERRORS
90000 CONTINUE
      CALL NCPRCV(STA,PID,LEN,MSG) !WAIT FOR COMMAND.
      IF(MSG(1).EQ.'SUBP') GO TO 90010
      IF(MSG(1).EQ.'SDBP') GO TO 90020
C      ?ERRORS  "INVALID DDBMS PROTOCOL CODE"
      GO TO 90000
90010 CONTINUE
      CALL DDBBOP(STA) !START-UP B/E PROCESS.
C      ?ERRORS
      CALL DDBBUT      !TRANSFER TO B/E USER TASK BLOCK
90020 CONTINUE
      CALL DDBBCL(STA)     !CLOSE DOWN B/E PROCESS.
C      ?ERRORS
      STOP
      END
```

Figure 4-11b. Sample Back-End Task Detail at Boston

```
      SUBROUTINE DDBBUT
      IMPLICIT INTEGER(A-Z)
      INCLUDE 'DDBBCT.COM'     !BACK-END TASK CONTROL
      INCLUDE 'ORSSDL.WRK'     !D/B UWA DEFINITION
      INTEGER POSN(20)
C     --- BACK-END USER TASKS ---
90000 CONTINUE
      CALL NCPRCV(STA,PID,LEN,MSG)
      IF (MSG(1).EQ.'SDBP')RETURN
      IF (MSG(1).EQ.'INVK') GO TO 90020
C     ERROR--INVALID DDBMS PROTOCOL CODE
      GO TO 90000
90020 TASKNO=MSG(2)
      CALL GETURL      !SET-UP STREAM FORMATS.
      CALL RTNACK('INVK+',LHPID,PID,2,MSG)
C
      GO TO (1,2),TASKNO     !INVOKE TASK TASKNO
C      ERROR--INVALID TASKNO
      CALL RTNACK('INVK-',LHPID,PID,2,MSG)
      GO TO 90000
C     **********  FE TASK 1 **********
1     CONTINUE  !NOT USED AT THIS HOST.
      GO TO 90000  !ENDFE 1 **********
C     **********  FE TASK 2 **********
2     CONTINUE
      POSN(2)='FIRST'
10002 CALL FNDUSB(ERR,'LINITM    ',POSN(2),'PARTS   ')
      IF(ERRSTA.EQ.307)GO TO 80002  !NO MORE IN SET.
      POSN(2)='NEXT'
      CALL FINDO(LINES)
      CALL GET(LINES)
      CALL GET(ORDERB)
      IF(.NOT.(OQTY .GE. 100))GO TO 10002
      CALL BLDCNK
      CALL SNDCNK
      GO TO 10002
80002 CONTINUE
      CALL SNDTAI
      GO TO 90000  !ENDFE 2 **********
C     --- END FE TASK BLOCK ---
      END
```

Figure 4–12a. Sample Back-End Task Control at Chicago

```
      PROGRAM BTSK12
      IMPLICIT INTEGER (A-Z)
C     BACKEND TASK 1 AT HOST 2, UNDER ORDRDG.
      INCLUDE 'DDBBCT.COM'      !BACK-END TASK CONTROL
C
      GHSTID='PHILA   '
      GSNAME='PASSDG  '
      LHSTID='CHICAGO '
C
      CALL DDBBSU(STA)       !ESTABLISH RETURN COMMUNICATIONS
      IF(STA.EQ.0) GO TO 90000
C       ?ERRORS
90000 CONTINUE
      CALL NCPRCV(STA,PID,LEN,MSG)  !WAIT FOR COMMAND.
      IF(MSG(1).EQ.'SUBP') GO TO 90010
      IF(MSG(1).EQ.'SDBP') GO TO 90020
C       ?ERRORS "INVALID DDBMS PROTOCOL CODE"
      GO TO 90000
90010 CONTINUE
      CALL DDBBOP(STA)
C      ?ERRORS
      CALL DDBBUT
C      CLOSE DOWN BACKEND PROCESS
90020 CONTINUE
      CALL DDBBCL(STA)
C      ?ERRORS
      STOP
      END
```

companion back-end task, described later, the necessary find owner access to get the required ORDERB data from the LINITM confluency base record is accomplished. The 10001 loop and GETCNK call deliver the required data to the UWA. The user body at this level consists of the WRITE statement.

In the DDBSTT sub-routine the ULIST/RLIST arrays are initialized using information derived from parsing the :USING/:RETURNING clauses or equivalent analysis. These arrays are used to establish the stream formats.

4.3.2 BTSK11 Program Description

Program BTSK11 is the back-end support program for task 1 at host 1 (BOSTON). It consists of a main program and a subroutine DDBBUT, containing the back-end user tasks for this transaction at this back-end host. Aside from some data variable initialization, all back-end main programs are the same. If the variable initialization is moved to the beginning of the back-end user task sub-routine and properly initialized from the main program, all back-end main programs can be made the same.

Figure 4–12b. Sample Back-End Task Detail at Chicago

```
      SUBROUTINE DDBBUT
      IMPLICIT INTEGER(A-Z)
      INCLUDE 'DDBBCT.COM'      !BACK-END TASK CONTROL
      INCLUDE 'PASSDL.WRK'      !D/B UWA DEFINITION
      INTEGER POSN(20)
C     --- BACK-END USER TASKS ---
90000 CONTINUE
      CALL NCPRCV(STA,PID,LEN,MSG)
      IF (MSG(1).EQ.'SDBP')RETURN
      IF (MSG(1).EQ.'INVK') GO TO 90020
C     ERROR--INVALID DDBMS PROTOCOL CODE
      GO TO 90000
90020 TASKNO=MSG(2)
      CALL GETURL      !SET-UP STREAM FORMATS.
      CALL RTNACK('INVK+',LHPID,PID,2,MSG)
C
      GO TO (1,2),TASKNO
C      ERROR--INVALID TASKNO
      CALL RTNACK('INVK-',LHPID,PID,2,MSG)
      GO TO 90000
C     **********  FE TASK 1 **********
1     CONTINUE
      POSN(1)='FIRST'
10001 CALL FNDUKB(ERR,'PART',POSN(1),'PARTNO')
      IF(ERRSTA.EQ.307)GO TO 80001  !NO MORE CALC RECORDS.
      POSN(1)='NEXT'
      CALL BLDCNK
      CALL SNDCNK
      GO TO 10001
80001 CONTINUE
      CALL SNDTAI
      GO TO 90000  !ENDFE 1 **********
C     **********  FE TASK 2 **********
2     CONTINUE
      GO TO 90000 !ENDFE 2 ***********
C     --- END FE TASK BLOCK ---
      END
```

However, with the current structure, the main program initialized the global host identification (PHILA), the global schema name (ORSSDG), and the local host identification (BOSTON). The DDBBSU routine establishes return communications with the front-end. The 90000 loop is the command processing loop. Control waits at the message receive, NCPRCV. At this point the only valid actions are to start-up a back-end process (SUBP) and to shut-down a back-end process (SDBP).

To start-up a back-end process, the DDBBOP module is called. This routine opens the local database. Control is then passed to the back-end user tasks by calling routine DDBBUT. A back-end process is shut-down by receiving the shut-down command or by returning from the DDBBUT routine.

The back-end user tasks are found in the DDBBUT subroutine. Descriptions of the back-end control task common and back-end database UWA descriptions are copied into the program source with the FORTRAN INCLUDE statement. The POSN array is used to control FIRST NEXT access to sets of records. Once activated, the DDBBUT routine waits for a message at label 90000. Normally, an invoke task (INVK) is received. In this case, the FE task number (TASKNO) is identified, the stream format is defined, (GETURL), and a return acknowledgement is sent (RTNACK). Control is then transferred to the specific FE task by the computed go to.

Only back-end FE task 2 is relevant to this host. The 10002 loop and FNDUSB call define the set of records for this FE task. When the underlying record set is exhausted, ERRSTA 307 is returned, causing control to be transferred to 80002. At 80002 the stream tail is sent (SNDTAI). Within the body of the FE loop the necessary set owner accesses and record gets are performed.

Since the FE statement included a record filter condition, (OQTY .GE. 100), that condition is used to exclude a base record instance from stream inclusion by a SEEDFE generated conditional branch to 10002. This condition test can include any variables with values at that point, including data derived in the loop body. The call to BLDCNK moves the data from the UWA to the message buffer. SNDCNK sends the chunk to the front-end.

4.3.3 BTSK12 Program Description

Program BTSK12 is the back-end program for task 1 at host 2 (CHICAGO). It consists of a main program and sub-routine DDBBUT. Since all back-end programs have the same structure, only the specific differences in the FE task sub-routine are described.

The 10001 loop and FNDUKB retrieve the relevant record set using keys. There is no applicable filter condition. Since CALC duplicates were not allowed, it is not necessary to generate the loop; the GO TO 10001 need not be generated.

4.4 Multi-Record Data Streams

The higher-level language discussed in this chapter is a set-level language; however, the multiple-key retrieval form of the "FOR EACH" statement and the recursive form of the "FOR EACH" statement do represent procedures more powerful than CODASYL set-level access. If the implementation of the SEEDFE compiler can recognize adjacent FE loops whose target records are collocated, further gains in distributed performance are possible.

Consider a request for order information (ORDERB and associated LINITM records) against the schema of figure 4–3. This transaction to retrieve order information is represented by two embedded FOR EACH loops:

```
FE ORDERB
    FE LINITM

    body to process order information

    ENDFE
ENDFE
```

The set level decomposition of this two-loop request results in two FE tasks, one for each FE loop. The consolidation of FE tasks improves system performance by reducing the communications requirements. Fewer, but larger, messages are sent. The more intelligent decomposition recognizes the collocation of the ORDERB and LINITM records and generates one FE task for the two original loops. Such actions represent a shift towards viewing SEEDFE as a non-procedural specification rather than a strictly procedural language.

The internal changes to accomplish such a consolidation during decomposition are straightforward, but the control structure and internal data format are more complicated. Data from multiple records must now be manipulated.

Handling multiple-record data in a stream environment yields *hierarchical streams*. In the order example, a stream would contain data from an ORDERB record followed by data from LINITM records. If information on multiple orders was selected, the stream would contain subsequent ORDERB-LINITM data. The mechanism used to define stream formats must reflect this hierarchical structure. The stream fragments themselves can be coded to reflect their source record type. A portion of the hierarchical stream which could have been generated from ORDERB-LINITM data is illustrated in figure 4–13. In this figure a variable length format is used for the stream. A dot (.) is used as a field deliminator. A slash (/) defines the end of data derived from a record instance. An .EOC. defines the end of a hierarchical segment. An .EOS. defines the end of the stream.

The above format is arbitrary and is used for illustration. An alternative fixed length field format could have been used. In either case, the sender and receiver agree on a format for the stream data. In the illustrated case, the

Figure 4–13. Hierarchical Stream Example

```
.ORDER.11111.JONES,INC.5-AUG-79./
.LINE.01.5.LA36.$1200./
.LINE.02.5.DL11.$500./
.LINE.03.1.VT100.$1500./
.EOC.
.ORDER.22222.SMITH,INC.10-SEPT-79./
.LINE.01.2.LA36.$1200./
.LINE.02.1.VT52.$950./
.EOC.
 ...
.EOS.
```

variable-length field format was LINITM records within ORDERB records. ORDERB records included order number, customer name, and date; and LINITM records included line number, part number, and price. This format information can be sent in the header for the stream. Alternatively, each portion of a stream can be self-encoding.

4.5 Using External Schemas To Represent Transactions

Instead of using a high-level language specification of the program task against a global schema, an external schema [Clemons 76, Clemons 78a, Clemons 79] could be defined against the global schema. An external schema is a non-procedural definition of a tabular structure containing data from a database relevant to a specific application problem. The use of external schemas provides several advantages, including easier application programming, because all the navigation is already built into the external schema, and greater data independence.

Figure 4–14 contains an external schema definition sufficient to allow processing of the sample order-parts transaction. In this external schema a hierarchical data table, PART-USAGE-DATA, is defined. This table has a fixed length portion including the following fields: PARTNO, PDESC, and NUMBER-OF-REFERENCES. The PARTNO value "12345" identifies the relevant part for which the table will be constructed. The PDESC field is derived from the underlying PART record. The NUMBER-OF-REFERENCES is derived from counting the number of underlying LINITM records associated with part "12345."

PART-USAGE, a fixed-length entry which forms the basis of a repeating group, has three fields: LINENO, OPRICE, and ORDERNO. PART-USAGE data is derived from the underlying LINITM record instances

Figure 4–14. Order-Parts External Schema Definition

```
EXT-SCHEMA PART-USAGE OF SCHEMA ORDERDB.

01  PART-USAGE-DATA.
    03  PARTNO  WHERE PARTNO = "12345"
    03  PDESC.
    03  NUMBER-OF-REFERENCES  COUNT OF LINITM  SUMMARY.
    03  PART-USAGE  OCCURS DEPENDING ON NUMBER-OF-REFERENCES
                      MAXIMUM OF 500
                      OPTIONS COMPLETE ALL
                      SORT BY CUSTNO, ORDERNO
                ACCESS LINITM OWNED BY PART IN SET PARTS.
        05  LINENO.
        05  OPRICE   ACCESS ORDERB IN SET LINES.
        05  ORDERNO  ACCESS ORDERB IN SET LINES.
```

associated with set PARTS owned by the "12345" part record instance. OPRICE and ORDERNO values are derived using find owner accesses in set LINES from the LINITM record instance. The PART-USAGE data includes data from all relevant LINITM records, not to exceed 500 instances, and is sequenced by ORDERNO within CUSTNO.

Instead of writing an FE program, the application programmer invokes an external schema. A program to do this is illustrated in figure 4–15. The READ statement causes an instance of the data structure defined by the external schema to be constructed and moved to a work area accessible to the programmer.

The same logic, described earlier in this chapter to decompose a high-level FOR EACH program into distributed tasks, can be applied to the software which assembles the instances of the external schema at execution time.

Figure 4–15. Application Transaction Using External Schema

```
MAIN-LINE.
   OPEN EXT-SCHEMA PART-USAGE.
         .
         .
   READ PART-USAGE-DATA.

   PERFORM LINE-PRINT NUMBER-OF-REFERENCES TIMES.

   CLOSE PART-USAGE.

   STOP RUN.
LINE-PRINT.
   DISPLAY PARTNO,PDESC,LINENO,OPRICE,ORDERNO.
```

4.6 Summary

In this chapter, the overall architecture of DSEED, a top-down DDBMS based on CODASYL-like structures, was described. The close relationship between the SEEDFE high-level data language, the global schema mechanism, and the implementation environment was discussed. A description of the process was given by which a transaction, represented as a FOR EACH program, could be translated by the SEEDFE pre-processor into front-end and back-end tasks with all the details of interprocess communications to support data transfer handled by the system. The process was illustrated for a distributed transaction against an order-parts database. The details of the DDBMS protocol used to transport data and its implementation in support procedures to define stream communications were described.

As was illustrated in this chapter, there is a close relationship between the data language and the programming language, and there are advantages to viewing them as integrated. The automatic generation of the using and returning information could result from a very detailed parse of the program, a task which a compiler must perform to generate its code. At some point compiler tasks and database tasks must be integrated. Until this occurs the only solution is to use a pre-processor. Since existing data languages have not as yet offered the full range of computational and procedural capabilites required by everyday transaction processing, existing programming languages must be used to define transaction processing.

5

Performance Modeling of Distributed CODASYL Structures

5.0 Introduction

The implementor of DDBMS software can choose from a large number of possible software architectures. Although there are many important operational characteristics to consider when choosing an architecture, performance issues are a central factor. Analytic performance models, developed to assist the system implementator in this difficult comparison, are described.

- Is the use of a high-level data language, parallelism, or data redundancy ever indicated?
- What happens when communications costs are high or low?
- What effect does a change in the retrieval versus update mix have on architecture choice?

5.1 The Four Architectures Modeled

Four architectures or strategies have been identified for distributing CODASYL-like structures.

- *Remote Area Model (RAM)* — In this model the database areas are distributed as remote files. Requests to read or write file records (CODASYL pages) are transmitted in the network and data pages are transmitted in return.
- *Remote Database Model (RDM)* — In this model independent database management systems operate at each host and DML level commands and data records are transmitted. A global view mechanism is used to hide the location of the individual databases and to

provide geographic transparency. Currently, each DBMS is a CODASYL system.

- *High-Level Language Model (HLL)* — In this model a high-level DML is used to communicate with the remote databases. A set-level language description, which is transmitted as a single command, is used to define the required data, which is transmitted as a single message. As in the RDM model, a global view mechanism is used to provide geographic transparency.
- *Parallel Processing Model (HPP)* — In this model the HLL model is extended to support the polling in parallel for the existence of transaction entry-level records. In contrast, the HLL model supported the sequential polling of hosts for these records.

All models are extensions of the Integrated Database Model [Gerritsen 77a]. Each model uses a given global schema structure and transaction mix to minimize operational costs, which include disk costs, cpu-memory costs, and communication costs. Although formulated as optimization models, only simple heuristics, implicit enumeration, and decision support facilities are used at this time to arrive at solutions. The nature of these models and the underlying distribution strategies are summarized in this chapter; the development of equations for each model can be found in appendix A.

5.2 Overall Structure of the Models

The information requirements of the models are discussed in this section. Each model takes a given transaction structure and associated usage pattern information, together with schema structure and hardware characteristics, to produce a set of optimal design decisions at associated minimum cost.

5.2.1 Model Inputs

The model inputs include: hardware parameters, including communications characteristics; database logical structure, represented by a global schema; and transaction information, including access path and usage patterns. Based on this environment, certain database decisions (data location and local structure decisions) are made to minimize operational costs. Operational costs include local disk storage costs, local CPU-MEMORY usage charges, and communications costs.

The transaction mix is represented by transaction structural characteristics and a usage pattern. The structure of a transaction is represented by its access path, the series of sets accessed to get to the target record once the entry-record type has been established. Access to a transaction entry-record is indicated to be "for all" of a given target record type or "for one" identified by key equality. A transaction's usage pattern is represented by the number of times the transaction is run from each host in a given time period.

Schema logical structure includes the length of data records, their origin hosts(s), and record-set structure. An indication whether each record type in the transaction is added or updated is included.

Hardware parameters include the cost of disk block storage, CPU-MEMORY usage, and the cost of communications, a function representing the cost of sending "x" characters from host A to host B. A similar function representing communications time is also used.

5.2.2 Communications Cost and Time Functions

The performance models summarized in this chapter use functions to determine the cost and time of sending messages in the distributed environment. Ctime (from, to, length) represents the time to send a message. Ccost (from, to, length) represents the cost of sending a message. Since both Ctime and Ccost are strongly dependent on message length, Ctime and Ccost are roughly proportional.

5.2.2.1 The cost function. The cost function definition is dependent solely on administrative policy. The problems of developing equitable charging schemes for networks are similar to those experienced by systems managers developing computer charging schemes. Usually only commercial data utilities take the time to develop charging policies for their network services.

In order to estimate Ccost in the non-local networks, TYMNET rates were used. Local and bus networks are less costly, requiring mostly the amortization of one-time hardware installation and software license costs. On-going costs include hardware/software maintenance and administrative costs. To represent an extreme case, local and bus network costs can be assumed to be zero.

The cost function typically includes two components: a charge per message sent and a cost per character transmitted. The number of characters sent is increased because of transmission protocol overhead. The increase is modeled by an expansion factor equal to 1.05, estimated from the DDBMS protocol described in chapter 4. The cost equation then becomes:

Ccost(ht,ht,len) = len * 1.05 * cost-char(hf,ht) + fixed-cost.

5.2.2.2 The time function. The time function definition depends upon the specifics of the communications protocol, the propagation time, the individual message overhead, and network congestion. The time function can be determined empirically by measurement or estimated using queuing-based models. In either case the "effective" process-to-process communications rate must be evaluated. A significant portion of the delay in process-to-process communications occurs in the local operating systems and not in the actual communication itself.

In order to estimate Ctime, a packet-type communications environment was assumed. Commercial data utilities are reluctant to release performance information, or they quote unrealistic data. For these reasons, ARPA-NET data [Kleinrock 74, Mamrak 77] was used to develop a simple equation for Ctime in the non-local region of interest.

The time function for a packet-network includes three components: a local-host delay, and en route delay, and a propagation time. The local-host and en route delays are assumed, for simplicity, to be constant and equal. The en route delay is the service time in an intermediate node multiplied by the number of nodes traversed going from hf to ht. The transmission time is dependent on the baud rate and expanded message size. The time equation becomes:

Ctime(hf,ht,len) =
2 * (local-host-delay) +
number-of-nodes-traversed(hf,ht) * svc-time +
transmission-time(len).

5.2.3 Communications Parameters

Four communications speeds will be evaluated: 300, 1,200, 9,600, and 56,000 baud. In each case, both "no cost" and "full cost" communications will be evaluated. The "full cost" communications include a cost-per-byte of $0.00010 for 300 baud and $0.00003 otherwise. These costs are based on TYMNET costs in the 300 to 2400 baud range. To reflect some amortization of fixed communications costs (fixed monthly charges or capital investment), a cost-per-message was also used. In all "full cost" communications, a cost-per-messages of $0.01 was used. A local-host-delay and svc-time of 1 milli-second was used. A fully connected network, where each host is connected to all other hosts, was assumed. This yields a number-of-nodes-traversed matrix of all zeroes.

5.2.4 Model Decisions

All models share the following local data structure decisions:

1. record location (via set or calc location)
2. set implementation,
3. record membership in singular set,
4. number of processing buffers at each host, and
5. fullness ratio (o.6 to 1.0) in each area.

The basic unit of distribution is the CODASYL record, except in the remote area model, where the area (file) is the unit of distribution. Instances of the unit of distribution are assigned to a host.

An additional decision, not present in the remote area model, is the record location mode binding time decision. The decision of where to locate record instances can be made at schema definition time or execution time. If all records of a given type are stored at a single host determined at execution time, the record location mode decision is *static*. If the record location mode decision depends upon execution time characteristics, the decision is *dynamic*. Two types of dynamic mode can be considered. The term *dynamic-local* will refer to the situation where all records originating at a site are stored locally at that site. The term *dynamic-mobile* will refer to the situation where a record's location is specified at execution time by the user or the system. Current model formulations do not support the dynamic-mobile location mode because of the increased complexity (distributions and a stochastic analysis would be required).

5.3 Distributed Order-Parts D/B Example

5.3.1 Order-Parts Example Schema Structure

A sample database structure, based on the schema of figure 4–4, was chosen and six typical transactions were defined. Order information (ORDERB-LINITM) can originate at each host in our hypothetical network of four hosts (BOSTON, CHICAGO, SAN FRANCISCO, and SEATTLE). Part information (PART) originates at the company's centralized warehouse in

CHICAGO. The record sizes used for the ORDERB, LINITM, and PART records were 120, 50 and 200 characters, respectively. The record origin frequencies for each record at each host are: (300, 600, 500, 200) for record ORDERB; (9000, 18000, 15000, 6000) for record LINITM; and (0, 500, 0, 0) for record PART.

The set LINES is not distributed, i.e., all LINITM records of a given instance of the LINES set are at one location. In contrast, the set PARTS is distributed, i.e., LINITM records in a single instance of set PARTS can be found at locations different from the PART owner instance.

5.3.2 Order-Parts Example Transaction Set

The test mix of transactions included the following six transactions:

1. "Add an order"—add a new ORDERB record.
2. "Add a line to an order"—get ORDERB record, add new line.
3. "Check part information"—get part record and display all information.
4. "Update part inventory"—get part record and update qty-on-hand.
5. "Check part usage"—list all orders for all parts (PART to LINITM to ORDERB).
6. "Print all local orders"—print ORDERB and associated LINITM information for all orders at a given location.

Each of these transactions can be run from each host. The run frequencies for a given month from each host for these transactions are assumed to be: (30,60,50,20) for transaction 1; (10,20,5,4) for transaction 2; (60,20,30,10) for transaction 3; (0,600,0,0) for transaction 4; (0,20,0,0) for transaction 5; and (20,20,20,20) for transaction 6. No minimum processing time constraint was placed on any transaction.

5.4 Specific Model Formulations

In this section the overall nature of each performance model is summarized. The actual equation development has been placed in appendix A to allow a concise summary to be developed here.

5.4.1 The Remote Area Model

In this architecture each area is assigned to a host. Access to an area remote from the host of the user incurs an additional overhead due to the transmission of the data block request and associated return transmission of the data block. Because data distribution is handled almost exlusively in the file system, this architecture is fairly easy to implement. However, with the large communications delays and costs, which are typically associated with geographically dispersed networks, this architecture should exhibit poor performance characteristics. This architecture may perform well where the distributed hosts are geographically close, allowing high bandwidth communications.

This model is an extension of the Integrated Database Model [Gerritsen 77a]. Instead of record-to-area decisions, record-to-host decisions are made. All records at a given host are assumed to be located in the same area. The number of buffers and the amount of free space at each host can be set independently.

5.4.2 The Remote Database Management System Model

The Remote Database Management system model is based on the concept of back-end database machines, but is augmented by global schema mechanisms to ease the programmer's burden. DML commands are passed to the required back-end database machines from the location where the transaction is run.

5.4.3 The High-Level Data Language Model

The High-Level Data Language model is a series of adjacent or embedded FOR EACH . . . ENDFE constructions, a loop construction representing processing of all records of a given type in an area-set, calc-set, or member-set. The record selection clause of the FOR EACH reduces the number of member records which must be sent back to a requestor. Processing includes the application of system functions (count, min, max, sum, average) or user procedure to the retrieved records.

In order to capture the behavorial characteristics of a High-Level Data Language program, the following information must be estimated for each transaction. First, for each record-type accessed along the access-path, what percentage of the data record is actually required (derived from items referenced)? Second, after the application of set-domain functions, how many characters need to be returned? And finally, what percentage of the records pass the qualification test?

5.4.4 The HLLP Model With Parallel Processing

In this model the high-level language model is extended to include uses of parallel processing. At this time only the effects of parallel polling to find an entry-record are modeled. Although the parallel processing of distributed set components is possible, this modeling was not done. The estimation of parallel processing time as the longest of the parallel tasks seemed to be an overstatement of the benefits of parallel processing.

5.5 Solution Procedures

The models, as currently formulated, are complex nonlinear, 0 1 integer programming problems. Although sophisticated O.R. techniques may someday lead to efficient solution procedures [Jaikumar 78, Fisher 78, Hochbaum 78], our current efforts have been directed towards using simple heuristics, implicit enumeration, and decision support facilities. If the models are used to support a database administrator in a distributed environment, the need for efficient solution procedures becomes more critical. This claim is made because of the ongoing nature of the database administrator's work, which requires constant reevaluation of his database designs. Since a one-time comparison of architectures was attempted here, less sophisticated solution techniques could be used. However, this does not imply that the distributed system software architect would not benefit from better solution procedures.

The solution program is a modular program written in FORTRAN. The program reads the schema structure and transaction information during its initialization phase. Transaction access path frequencies are converted to records-set access path frequencies by the transaction data-entry routine. A two-stage approach is used. First, the data are distributed (a starting set of database decisions is chosen); and second, the local database decisions can be reoptimized. Since the major costs were communications and disk costs (mostly to store the data), this reoptimization step was not performed.

The data location decisions are enumerated. Where required, dynamic-local location mode is treated as a pseudo-location, identified by the number of hosts plus one.

Within these loops the record location mode decisions are enumerated. The via constraint, that a record can only be via a set which it is in, is checked at this time. The other via constraint, applicable to the RAM model, that the owner and member record types of a via set must be collocated, is also checked at this time. Because the other models support the notion of distributed sets, this constraint is not appliable to them.

The other database decisions were set once and were not changed during this phase. Specifically, sets are implemented with chain-owner pointers, entry-records are in singular sets, each area is 80 percent full, and four buffers are allocated at each host.

Since the communications costs influence the results more than disk costs and cpu-memory charges, record location is viewed as the critical decision. Once records are located, the database decisions can be reoptimized at each local database. Alternatively, heuristics can be used to establish initial or final database decisions settings. For example, the number of buffers required at each host should be set to the average or maximum size of the independent via clusters traversed by the transaction. Once the record calc-via decisions are set, the model evaluation routines are called.

5.5.1 Enumeration Complexity

Setting the database decisions (local record location, set implementation, singluar set membership, number of buffers, and page size) by default or heuristics, leaves the record location and location mode decisions. Since these decisions are inter-related, the problem is a nonlinear, 0–1 integer minimization problem with transportation structure [Garfinkel 72]. With our simplified evaluation scheme, the number of enumeration passes required, defined using the variables defined in appendix A, is given by:

$$\text{(No. passes)} = (w+1) ** m * \mathbf{X}_i (1 + \Sigma_j (M_{ij} + 1)).$$

The first product factor represents the location possibilities for each record. There are m record types. Each record can be located at w locations plus one location for the pseudo-location representing dynamic-local location mode. The second product factor represents the location mode possibilities for a record. Each record can be located CALC or via a set of which it is a member. For our order-parts example, 375 passes are required.

5.6 Results for Distributed Order-Parts D/B Example

The results for the order-parts database example are summarized in figures 5–1, 5–2, and 5–3. Figure 5-1 summarizes the optimal decision settings and associated costs for the "no cost" and "full cost" communications cases. In this figure the optimal decision vector has been translated from the underlying 0–1 decisions to terminology understood by a DBA. For each record type, the location mode is indicated as Sn, static at host n; or DL,

Figure 5–1. Optimal Decisions and Associated Costs

Baud	Model	Min-Cost$		Optimal Decisions	Comm.$	CPU-MEM$
				Rec. 1; Rec. 2; Rec. 3 ORDERB; LINITM; PART		
300	RAM	6698		S4,calc;S2,via2;S2,calc	0	6122
300	RDH	676		S2,calc;S2,via1;S2,calc	0	100
300	HLL	642		S2,calc;S2,via1;S2,calc	0	66
300	HPP	642		S2,calc;S2,via1;S2,calc	0	66
1200	RAM	2137		same as 300 baud	0	1561
1200	RDH	607		same as 300 baud	0	31
1200	HLL	597		same as 300 baud	0	21
1200	HPP	597		same as 300 baud	0	21
9600	RAM	807		same as 1200 baud	0	230
9600	RDH	586		same as 1200 baud	0	10
9600	HLL	584		same as 1200 baud	0	8
9600	HPP	582	*	S2,calc;DL,via1;DL,calc	0	6
56K	RAM	649		same as 9600 baud	0	73
56K	RDH	584	*	S2,calc;DL,via1;DL,calc	0	8
56K	HLL	582	*	S2,calc;DL,via1;DL,calc	0	6
56K	HPP	580		same as 9600 baud	0	4
300	RAM	8150		S4,calc;S2,via2;S2,calc	1452	6122
300	RDH	734		S2,calc;S2,via1;S2,calc	58	100
300	HLL	659		S2,calc;S2,via1;S2,calc	16	66
300	HPP	659		S2,calc;S2,via1;S2,calc	16	66
1200	RAM	2800		same as 300 baud	663	1561
1200	RDH	653		same as 300 baud	46	31
1200	HLL	606		same as 300 baud	8	21
1200	HPP	606		same as 300 baud	8	21
9600	RAM	1469		same as 1200 baud	663	230
9600	RDH	632		same as 1200 baud	46	10
9600	HLL	593		same as 1200 baud	8	8
9600	HPP	593	*	same as 1200 baud	8	8
56K	RAM	1312		same as 9600 baud	663	73
56K	RDH	630	*	same as 9600 baud	46	8
56K	HLL	591	*	same as 9600 baud	8	7
56K	HPP	591		same as 9600 baud	8	7

[Different decisions between "full cost" and "no cost" communications at same baud rate and model indicated by *]

Figure 5-2. Unit Transaction Execution Times, seconds

"No	Cost"	T1	T2	T3	T4	T5	T6
Baud	Model	ADD-O	ADD-L	CK-PRT	UP-PRT	AL-P-O	ALL-L-O
300	RAM	82.5	22.9	21.3	.23	4491.	3480.
300	RDH	3.3	3.8	6.5	.16	14.4	55.5
300	HLL	2.7	3.5	.67	.08	14.4	39.7
300	HPP	2.7	3.5	.67	.08	14.4	39.7
1200	RAM	20.8	5.8	5.4	.23	1164.	882.
1200	RDH	1.0	1.14	1.7	.16	14.4	14.2
1200	HLL	.74	1.06	.23	.08	14.4	10.2
1200	HPP	.74	1.06	.23	.08	14.4	10.2
9600	RAM	2.8	.80	.75	.23	194.	124.
9600	RDH	.33	.36	.29	.16	14.4	2.2
9600	HLL	.17	.35	.10	.08	14.4	1.6
9600	HPP *	.17	.46	.04	.03	10.0	1.5
56K	RAM	.68	.21	.20	.23	79.4	34.0
56K	RDH *	.25	.22	.21	.29	9.9	.67
56K	HLL *	.10	.25	.19	.19	9.9	.52
56K	HPP	.10	.25	.01	.01	9.9	.52

"Full	Cost"	T1	T2	T3	T4	T5	T6
Baud	Model	ADD-O	ADD-L	CK-PRT	UP-PRT	AL-P-O	ALL-L-O
300	RAM	82.5	22.9	21.3	.23	4491.	3480.
300	RDH	3.3	3.8	6.5	.16	14.4	55.5
300	HLL	2.7	3.5	.67	.08	14.4	39.7
300	HPP	2.7	3.5	.67	.08	14.4	39.7
1200	RAM	20.8	5.8	5.4	.23	1164.	882.
1200	RDH	1.0	1.2	1.7	.16	14.4	14.2
1200	HLL	.74	1.1	.23	.08	14.4	10.2
1200	HPP	.74	1.1	.23	.08	14.4	10.2
9600	RAM	2.8	.80	.75	.23	194.	124.
9600	RDH	.33	.36	.29	.16	14.4	2.2
9600	HLL	.08	.36	.10	.08	14.4	1.6
9600	HPP *	.17	.35	.10	.08	14.4	1.6
56K	RAM	.68	.21	.20	.23	79.	34.
56K	RDH *	.25	.27	.12	.16	14.4	.71
56K	HLL *	.10	.26	.09	.08	14.4	.54
56K	HPP	.10	.26	.09	.08	14.4	.54

Figure 5-3. Unit Transaction Execution Costs, dollars

"No Cost"		T1	T2	T3	T4	T5	T6
Baud	Model	ADD-O	ADD-L	CK-PRT	UP-PRT	AL-P-O	ALL-L-O
300	RAM	1.31	.36	.34	.00	71.40	55.33
300	RDH	.05	.06	.10	.00	.23	.88
300	HLL	.04	.06	.01	.00	.23	.63
300	HPP	.04	.06	.01	.00	.23	.63
1200	RAM	.33	.09	.09	.00	18.51	14.02
1200	RDH	.02	.02	.03	.00	.23	.23
1200	HLL	.01	.02	.00	.00	.23	.16
1200	HPP	.01	.02	.00	.00	.23	.16
9600	RAM	.04	.01	.01	.00	3.09	1.97
9600	RDH	.01	.01	.00	.00	.23	.03
9600	HLL	.00	.01	.00	.00	.23	.02
9600	HPP *	.00	.01	.00	.00	.16	.02
56K	RAM	.01	.00	.00	.00	1.26	.54
56K	RDH *	.00	.00	.00	.00	.16	.01
56K	HLL *	.00	.00	.00	.00	.16	.01
56K	HPP	.00	.00	.00	.00	.16	.01

"Full Cost"		T1	T2	T3	T4	T5	T6
Baud	Model	ADD-O	ADD-L	CK-PRT	UP-PRT	AL-P-O	ALL-L-O
300	RAM	1.47	.45	.42	.00	88.54	68.73
300	RDH	.07	.09	.14	.00	.23	1.50
300	HLL	.06	.08	.03	.00	.23	.76
300	HPP	.06	.08	.03	.00	.23	.76
1200	RAM	.40	.13	.12	.00	26.35	20.14
1200	RDH	.03	.04	.05	.00	.23	.73
1200	HLL	.02	.04	.02	.00	.23	.21
1200	HPP	.02	.04	.02	.00	.23	.21
9600	RAM	.11	.05	.05	.00	10.92	8.09
9600	RDH	.02	.03	.03	.00	.23	.53
9600	HLL	.01	.03	.02	.00	.23	.08
9600	HPP *	.01	.03	.02	.00	.23	.08
56K	RAM	.08	.04	.04	.00	9.09	6.66
56K	RDH *	.02	.03	.02	.00	.23	.51
56K	HLL *	.01	.03	.02	.00	.23	.06
56K	HPP	.01	.03	.02	.00	.23	.06

dynamic-local. In addition, local record placement is indicated as calc or via a specific set. For example, the 300 baud, RDH model, "no cost" result is optimal with the following decisions:

- record one—static at host 2 with calc location mode;
- record two—static at host 2 via set 1;
- record three—static at host 2 with calc location mode.

In other words, centralize the database at location two.

At the optimal decisions indicated by figure 5–1, there is an associated time to run each transaction, summarized in figure 5–2, and an associated cost, summarized in figure 5–3. Disk costs are $576 per month. During the month, 1019 transactions were executed. If the disk costs are prorated equally to each transaction execution, each transaction would be assessed $0.57 for disk charges.

A few remarks will be made about these results. First of all, the results are relevant only to the specific sample data structure; and moreover, the results are very dependent on the transaction mix and associated run frequencies. It is possible for one transaction or a few transactions to dominate the results. For example, T5 and T6 contribute substantially to the cost performance on a per transaction basis. Fortunately, they are only run once a work day. On the other hand, some transactions can have a negligible cost. For example, T4 costs less than one cent to execute.

As expected, the RDH model yielded better time and cost performance than the RAM model; the HLL model yielded better time and cost performance than the RDH model. The HPP model did marginally better than the HLL mode in some cases.

In the "no cost", i.e., no communications costs results, the RAM overall optimum cost performance was approximately 10 times worse than the RDH model for the sample transaction mix at 300 baud. At the other data rates, the RAM to RDH costs performance ratios were slightly better. In the "full cost" results, the RAM overall optimum cost performance was approximately 11 times worse than the RDH model for the sample transaction mix at 300 baud. At the other data rates, the RAM to RDH optimum cost performance ratio ranged from 4:1 to 2:1.

Changing baud rates did reduce transaction times. This reduction was proportional to the increased transmission speeds, although a doubling of rate did not double the time performance. This was particularly noticable at the higher data rates.

When the optimal decision points are reviewed, it is found that the dynamic-local decision was selected in some of the models. Surprisingly, the decision to centralize the database was also selected in some cases, e.g., the "no cost" RDH model at 300 baud.

5.7 Remarks Concerning Modeling

5.7.1 Limitations of Modeling

As with all modeling efforts of this nature, certain objections can be raised. First, the amount of information required is large and expensive to capture. However, the schema information is available. Much of the transaction structure information can be derived automatically from well-designed transaction writing systems, e.g., the DSEED pre-processor, or defined manually as transactions are written. The run frequencies can be captured periodically in an ongoing system, which is an expensive operation, or can be estimated as part of systems analysis for new systems. The communication time function can be measured empirically. The communication cost function is dictated by policy, either external or internal.

Second, the mathematical structure of the models is complex. The structured approach, which the models define for us, improves our intuition about the problem. The process of defining the models forced us to think through the interactions between the key variables affecting system performance. Moreover, this insight helped in the prototype design. Also, the use of heuristics, which the models help us better understand, may yield operationally acceptable solutions.

Finally, the models have not been validated against live situations. At some point this should be attempted, not so much to verify the exact costs and times, but to see if the relative orderings of performance predictions hold true. Even without validation, however, the models offer a starting point for the system implementor and database administrator. Despite the objections raised, development of the models has sharpened our insight into some of the problems of distributed structures.

5.7.2 Recommended Model Changes

After working with the models for some time, the following changes in approach are recommended. First of all, transactions should be represented by their transaction path throughout the modeling process. Upon input, transactions are represented by their transaction path; however, this procedural representation is then converted to a nonprocedural representation. For example, a transaction to print an order is entered by its transaction path: get record ORDERB by key and then process all the LINITM records. This information is then converted to access frequencies involving the entry-record and the sets along the access path. It is preferable to account for the run frequencies after the full transaction time has been calculated. Also, instead of the constant reidentification of the owner and member of the set along the access path in the equations of the models, this information should be included in a transaction representation, which is defined once, and the models should be changed to use it. If these changes are made, the model formulations will be simplified. This building of a transaction representation can be done prior to the model processing phase, after the transaction path information is entered.

Our models currently do not exploit the potential of redundancy, nor do they reflect the costs. Once a decision is reached on how redundancy should be introduced into the architecture, changes to the model equations can be developed. However, the mathematical structure of the models is complicated by the addition of data redundancy, making the use of closed-form O/R techniques more difficult.

5.8 Summary

In this chapter, four strategies for the distribution of CODASYL-like structures (the remote area model, the remote database management system model, the high-level language model, and the parallel processing model) were studied using 0–1, nonlinear optimization models. The model decisions included data location and database implementation decisions. A hypothetical, but realistic, order-parts database structure and transaction mix was used as a sample problem and studied under varying communications services. Optimal solutions were found for the sample problem using a FORTRAN program which used a two-stage approach of setting initial database implementation decisions and then doing the data location. The constraint structure was used explicitly to minimize the number of cases which had to be examined. The relative ranking of the different models by their cost performance produced predictable results. The remote area model performed very poorly in the slow communications regions. The remote database model performed much better, 5 to 10 times better. The high-level language model and high-level with parallel polling offered further, but modest, cost performance improvements, with the parallel polling approach improving transaction times in the slower communications regions.

6

DSEED Distributed Update Control

6.0 Introduction

A number of distributed DBMS software design decisions were outlined in chapter 1. Up to this point, this study has stressed the importance of providing geographic independence for the application programmer. This was accomplished by using a multiple-schema architecture. This study has also investigated ways of achieving better performance in distributed database architectures. By using a high-level data access language, the effects of the distributed environment on performance were minimized. The transaction decomposition scheme, which integrated the multi-schema architecture with the high-level data language, provided geographic independence to the application programmer. The application programmer no longer concerned himself with the problems of data location, task decomposition, and interprocess communications.

In addition to these issues, distributed database software implementors must address the issue of distributed update control. Although no contributions are claimed by this study in this area, it is beneficial to explore how the DSEED architecture might address distributed update control issues.

In section 1, the implications of introducing data redundancy into the DSEED architecture are developed. In section 2, concurrent access control issues are discussed. In section 3, various operational issues which relate to recovery after system malfunction are addressed.

6.1 Data Redundancy in DSEED

The introduction of data redundancy into a distributed database structure can improve retrieval performance. However, the major advantage gained by the use of data redundancy in a distributed database is the improved system availability. If a host is unavailable, an alternate site with the required data can be accessed.

Although clear advantages are derived through the use of data redundancy, there are disadvantages. First of all, the introduction of data redundancy increases storage costs. Second, update procedures take more time because all copies of the redundant data must be changed. Moreover, because of the desire to maintain data consistency, recovery after failure during an update is more complicated. Partial changes must be "rolled-back." Also, updating redundant copies when some hosts are not available requires careful logging and subsequent processing to bring all copies of redundant data to a consistent state.

In the previous chapter the performance of four architectures for DDBMS were studied. The DSEED prototype architecture was based most closely on the high-level data language model. Data redundancy can be introduced into these architectures at different levels.

In the remote area model, data redundancy can be introduced at the database instance or area instance level, whereas the remaining architectures can additionally support redundancy at the record instance level. The existence of data redundancy impacts search strategies in a straightforward manner. If an area or database is replicated, the closest copy is used on retrieval and all copies are accessed on update.

If a record type is replicated, each individual location mode must be considered. If the record location mode is static, the policy described for area or database replication applies. If the record location mode is dynamic and the record type is a transaction entry-record, a strategy consistent with the nonredundant case is used. Namely, either a polling operation is used to find relevant record instances or a transaction entry-record directory, containing redundant entries, is searched. In either case, only one copy is actually accessed on retrieval requests.

When a record type which is a member of a set is replicated, the implementation of distributed sets is extended. The implementation of distributed sets, described in chapter 2, formed the basis of a distributed directory. The hosts which had member record instances for the set instance were indicated on the owner side of a distributed set instance. At each relevant host on the member side, a record instance, called a distributed set header, was created to own those member records of the distributed set instance at the host. There were as many distributed set header record instances for a distributed set instance as there were hosts having member instances of the distributed set instance.

If the member records of a distributed set are redundant, additional information is added to the owner side of the distributed set. Host pointers are added to indicate the redundant sites in addition to the primary site. Update processing to add set members requires that all relevant redundant sites be accessed. The retrieval processing of distributed set members involves only one of the possible hosts. For performance reasons a policy of picking a close host is preferable.

More complicated procedures to select which data locations to use have been proposed for some relational implementations [Epstein 78]. Because of the potential dynamic nature of record storage in DSEED and implementation of set directories as distributed structures, which are part of the underlying databases, run-time transaction optimization might not be possible in the DSEED architecture. In any event, it is possible to apply models similar to those developed in chapter 6 to optimally allocate data to minimize operational costs. Morgan and Levin discuss such allocation procedures for files [Levin 74, Levin 75, Morgan 77]. The dynamic-local location mode is based on the "locality of reference" assumption. If this assumption is not valid, then only static location should be used. If only static location modes are used, comparable optimizations can be performed. Also, it is always possible to separate the set directory structures from the underlying data structures. These structures can then be consulted independently of the underlying data to determine a processing strategy. Of course, by doing this, the advantages of the combined structure are lost. Additional disk accesses are required to maintain separate structures and additional communication time is required if the set directories are located at a host different from the distributed set owner and member records.

If data redundancy is used mainly to improve availability, a single "shadow" copy will likely be sufficient. The DSEED extensions described are sufficient to handle data redundancy of this kind. Although it is possible to extend the DSEED architecture to support extensive data redundancy and run-time optimization, the performance impact of doing so may not be acceptable. It remains to be seen if the benefits of such extensions in any architecture exceed the drawbacks of increased complexity.

6.2 Concurrent Access Control

Concurrent access control has two facets. The first facet deals with the locking of data which are undergoing update operations in order that other users of the data do not see an inconsistent picture of the data. The second facet deals with the prevention or detection of deadlock situations. The prevention of deadlocks is only possible when all required data can be identified beforehand or requested in a strict sequence. Since the actual data instances required may be dependent upon information supplied or determined at execution time, it is difficult to prevent deadlocks in this environment. (Of course, it is always possible to make two passes through the transaction: one to identify the data resources and a second to request the locks.)

As was discussed earlier, the major difficulty faced by implementators of distributed concurrency control schemes is the achievement of adequate performance. Because of the dispersed nature of some networks and the associated process-to-process communications rates, lock setting operations take time. For this reason, many new schemes have been proposed to handle concurrency control in the distributed environment [Bernstein 77, Garcia-Molina 79, Lien 78, Lin 79, Minoura 79, Menasce 78, Ries 79b, Rosenkrantz 78, Rothnie 77a, Stonebraker 79b]. As Ries points out [Ries 79a], the selection of a concurrency control algorithm and the choice of implementation parameters is application dependent. In other words, the nature of the application (degree of concurrent access to the same object, the frequency at which deadlock situations occur, etc.) impacts the performance of the algorithm.

The algorithms discussed by Rosenkrantz, et al. [Rosenkrantz 78], are compatible with the DSEED architecture and could be used to solve the concurrent access problem. Two protocols based on time stamps are described. One protocol gives preference to younger transactions, while the other gives preference to older transactions. Although preference to older transactions is clearly the "fair" policy, the associated protocol has the undesirable effect that a transaction may continually be restarted until it gets all the resources it requires. However, once it receives this resource set, it cannot be interrupted and will run to completion. Both protocols are decentralized since no global resource graph is maintained. Unfortunately, because no such graph exists, some transactions not involved in a deadlock situation may be restarted. A centralized scheme where one network node maintains a global graph and a hybrid scheme, which distinguishes between local and global transactions, could overcome this weakness.

6.3 Recovery Issues

Recovery after system failure is more difficult in a distributed environment. A remote host may not be accessible because of communications failures or local malfunctions. With current machine and network failure rates, the probability is finite that a portion of the network is not available at any given time. Assuming that properly configured networks with multiple routing paths can minimize the effects of communications failures, we will look more closely at the effect of a host failure. If host failures are independent events and each host can fail with probability F, the probability of a network of n hosts being fully operational is:

Network-Up-Probability = (1—F)**n.

For failure probability of .01 a 10-node network has a 0.9 chance of being fully operational, a 20-node network has a 0.78 chance and a 50-node network has a 0.61 chance. System implementors must develop procedures which work in an environment where all hosts are not simultaneously available.

The notion of atomic transaction is introduced to define processing actions which take a database from one consistent state to another. If failure occurs the DDBMS software must guarantee that either the transaction completes or that it has no effect on the database.

In DSEED, a SEEDFE program is an atomic unit. Either the program runs to completion, in which case changes to the database due to program execution are reflected in the database; or, the program or a subordinate task does not complete due to system failure, in which case any changes to the database due to program execution are "rolled-back." In order to accomplish such recovery actions, changes to individual databases are staged or journaled locally. When all local sites notify the controlling site that the required databases' changes are complete and are pending commit, the controlling site instructs all local sites to commit the pending changes to the local database.

The implementation of this "two-phase commit" process requires that each local DBMS have certain functionality. Each local database must have a journal capability which allows updates to be staged until a commit is requested. This information must be secure even if the local host crashes. Also, a mechanism must be defined to allow "crashed nodes" to bring themselves into synchronization with the rest of the system when they return to operation. This can be implemented most easily by requiring newly initialized nodes to interrogate a network information node for relevant recovery information. Procedures to support DDBMS operation in a network which is partially up and procedures to support recovery after failure are research areas requiring much additional work.

6.4 Conclusions

In this chapter, the notions of data redundancy, concurrency control, and recovery in the DSEED architecture have been discussed. Although no contributions were made in these areas, their discussion completes the DSEED architecture.

Conclusions, Significance, and Further Research

7.0 Introduction

In the preceding chapters several major facets of DDBMS implementation were addressed. The development of a DDBMS software architecture which provided efficient, geographically transparent access to a distributed database was a major goal of this research. In this chapter the significance of this work is summarized and several avenues of future research are identified.

7.1 Significance

We have looked at many strategies for the distribution of CODASYL-like structures in order to answer a fundamental question: Can CODASYL-like architectures be used as a basis for distributed databases? Instead of agreeing that CODASYL is not workable in the distributed environment and therefore should not be used, as opponents of CODASYL-like architectures contend [Stonebraker 79a], we extended CODASYL to overcome its weak points in this environment while at the same time retaining CODASYL's advantages.

We have described an architecture, based on CODASYL-like structures, but we have moved a considerable distance from those specifications. A prototype architecture has been defined. Portions of the prototype architecture were implemented to gain a better understanding of the implementation problems. Key features of this prototype architecture are summarized below.

A multiple schema architecture, which includes the DSEED schema structures, was proposed for the distributed environment. By using the proposed data definition facility to define global schemas in a top-down, homogeneous data model environment, the goal of providing geographic transparency for the application programmer is achieved. We also moved away from the low-level DML to a high-level FOR EACH language, which had some non-procedural aspects. Transaction tasks, represented in this high-level FOR EACH language, were decomposed by a pre-processor utility into front-end and back-end tasks, with all details of inter-task communications incorporated. This automatic decomposition of transactions, which helps provide geographic transparency for the application programmer, is a key contribution of this study.

In order to better understand the trade-offs in DDBMS software design, the performance of alternative methods of distributing CODASYL structures was studied with a series of analytic performance models. More work to extend the models and define better solution procedures needs to be done; nevertheless, the development of the models yielded insight, particularly in the use of high-level languages, which is reflected in the DSEED prototype architecture. The analytic performance models are also a start at defining tools for the systems designer and database administrator. With these tools the systems designer can better select DDBMS software architectures and the database administrator can automate portions of his task.

7.2 Further Research

The completion of the research reported here has given insight into several related areas which might be fruitfully investigated.

7.2.1 Multiple Schema Architectures Revisited

In chapter 2 a multiple schema architecture for distributed databases was proposed. This architecture included: a set of external schemas to interface to the application programmer; a global conceptual schema to define the logical structure of the database and provide geographic transparency; a global internal schema to provide structure implementation details and address geographic distribution; and a set of underlying, local schemas (external, conceptual, and internal) to define the local data structures. This architecture is general enough to support a number of enhancements, including limited, heterogeneous data model support.

7.2.1.1 Stream processors and multiple schema. Buneman, in the Functional Query Language (FQL) [Buneman 79], and Hayward, in its predecessor, Q [Hayward 78], discuss stream processors in a single machine environment. As Buneman and Hayward suggest, stream processors have applicability in the distributed environment. The main advantage offered by a stream processor in the distributed environment is the convenient programmer interface for the DDBMS implementor. The programmer need not be concerned by the details of the stream contruction.

The use of stream processors can be addressed at two levels in this architecture. At the low-level, stream processors can be used to implement the mechanics of stream processing. In this case, each local database has a conceptual FQL interface to the local database structure. By building FQL interfaces for multiple data models, the local internal schemas can be representative of multiple data models. FQL programs become the basis of internal interface communications. The global internal schema is a set of FQL functions to provide the necessary data streams to support the global conceptual schema. The global conceptual schema can be the DSEED data model, which provides essentially a CODASYL data view, or an extended data model, i.e., the functional data model [Buneman 79, Shipman 79] or an entity-relationship model [Chen 79].

To support stream processors in the distributed environment, the *stream combiner* function must be extended. A stream combiner is a function which joins streams of like objects. The stream combiner must consult directory services for the location of databases. It must also provide additional functionality regarding how streams can be combined.

Consider a stream of a single record type, e.g., ORDERB. Each local order database can yield a stream of ORDERB records. These can be merged into a single stream of orders in three ways: merge them in an arbitrary way, merge them in a key sequence, or merge them host by host. Buneman and Frankel discuss a basic stream combiner [Buneman 79]. This function merges streams database by database, i.e., host by host if FQL is extended to the distributed environment.

Hierarchical streams can be merged in a similar fashion by using their root records to control the sequence of the merge. In FQL a stream can be defined to represent a hierarchy of records. Each set of children in a hierarchy is a list or stream. Streams of streams are a representation of a hierarchical structure. If a stream of streams, each of which represents a stream of hierarchies derived from one host, is merged, the desired hierarchical final stream is produced.

If transaction processing, rather than query processing, is to be supported, the stream processor must support update. Much work remains to be accomplished to integrate update into FQL.

7.2.1.2 External schemas and multiple schemas. As was discussed in chapter 4, external schemas can provide an alternative representation of the application program function. Also external schemas can be used to define containers or the data transport mechanism for stream processors.

A *container* is like a stream except that it is filled all at once and then transmitted, whereas a stream is transmitted in pieces. Streams have the advantage of not requiring storage of the entire relevant data structure, only the head portion, and may allow more processing overlap. This is particularly important if parallel processing of distributed set components is to achieve its true potential. On the other hand, containers have the advantage of providing all the data at once. In some situations it may be desirable to have all the information before one proceeds. Moreover, containers require a reduced number of messages and acknowledgements by sending one large message.

To use external schemas at the local level, a local external schema is added between the local conceptual schema and global internal schema. External schema invocations, the request that a data table be constructed, form the basis of internal interface communications.

For an external schema facility to be more useful in the distributed environment, update facilities must be provided. Clemons has discussed some of the update anomalies present with external schema facilities [Clemons 77]. If the external schema includes the computation of virtual fields, these fields cannot be directly changed by the external schema user, because they do not exist. For example, consider the calculation of the number of lines on an order or the total value of an order. Both of these fields could be virtual fields of an external schema. The user cannot directly change these fields. However, they can be changed indirectly by changing the underlying data. As illustrated by this example, it is proposed that these update problems for this application of external schemas be overcome in an obvious way. Update of virtual fields will simply not be allowed.

Using external schemas for updates allows a single data table to contain all the information necessary to update a portion of the database. The information is then sent to the relevant site and the update is performed. To add an order, the ORDERB and associated LINITM records are packed into a table and sent. For transactions involving access to multiple databases, each step is represented by an external schema and associated *invocation*. Figure 7-1 illustrates external schemas which could have been used to define tables for the data transfer to the local databases. In effect, external schemas are used to define containers. Alternatively, external schemas could be used to define the transport mechanism for a stream language. If the local databases have external schema facilities, the building of stream generators/consumers is made much easier.

7.2.1.3 Bottom-up architectures. A top-down architecture was explored in this study. Because of the potentially broader applicability of the bottom-up architectures discussed in chapter 1, the investigation of extending concepts developed here to support a bottom-up architecture is desirable. Some of the

Figure 7-1. External Schema Definition of Order Container

```
EXT-SCHEMA UPD-ORDER-O OF SCHEMA ORDERL FOR COBOL UPDATE.

01 ORDER.
   03  CUSTNO  WHERE CUSTNO = "12345".
   03  CUSTNM.
   03  ORDERDT.
   03  NUMBER-OF-LINES  COUNT OF LINE-DATA.
   03  LINE-DATA  OCCURS DEPENDING ON NUMBER OF LINES
                  MAXIMUM OF 500
                  OPTIONS ORDER BY LINENO.
       05  LINENO.
       05  OQTY.
       05  OPN.
       05  OPRICE.
       05  PARTNO  VALIDATE ACCESS PART IN SET PARTS.



EXT-SCHEMA UPD-ORDER-P OF SCHEMA PARTDL FOR COBOL UPDATE.

01  PART.
    03  NUMBER-OF-PARTS  COUNT OF PART-DATA.
    03  PART-DATA  OCCURS DEPENDING ON NUMBER-OF-PARTS
                   MAXIMUM OF 500
                   OPTIONS ORDER BY PARTNO.
        05  PARTNO.
        05  DISTRIBUTED-SET-INDEX.
            07  PARIIO.    /*DATABASE KEY OF ORDER*/
            07  PARIIH.    /*AT THIS HOST*/
```

concepts will not apply; others will have to be redefined. The possiblity of allowing a limited, multiple data model environment can be addressed. It is desirable to gain an understanding of where the problems are in defining the bottom-up mappings and what difficulties are inherited because of given, fixed local structures. Can anything be gained by requesting structural changes in local structures in an otherwise autonomous local database? Is there a minimal functionality for local DBMS, whose existence makes the implementation of bottom-up architectures easier? For example, the existence of *commit verbs* in all local DBMS's supporting the global schema makes the development of bottom-up architectures which support update much easier. (A commit verb guarantees that all changes in an identified set of code are performed against the database, or if a failure occurs during this set of code, no changes are made in the database.) Without such functionality•it is much more difficult to guarantee consistent update.

7.2.2 Degree of Concurrent Update

A critical aspect of the application which determines its performance characteristic is the degree of concurrent access and update required by the application. It may be possible to look at various applications and classify them according to their concurrent access requirements, regarding simultaneous access to objects and the potential for deadlock. The level of granularity is important [Ries 79a] and it is desirable to perform this analysis at several levels of granularity, e.g., record instance, database page instance, and database instance. For example, an airline reservation system must deal with a lot of concurrent access to flight-related records near flight time. At other times it may not. A bank checking application does not appear to involve a large amount of concurrent access to individual account records. Understanding the degree of concurrent access present in an application will allow a better match of DDBMS architectures with applications.

7.2.3 Model Enhancements

In this study, 0–1, nonlinear optimization models were developed and used to study the performance of various software architectures. Their structure was nonlinear and computationally inefficient methods were used to evaluate their optimal solutions. It may be possible, with reformulation, to reduce them to quadratic form with transportation structure. The nonquadratic complications arise because of the way dynamic-local location mode is handled. An alternative formulation may remove these complications.

In any case, the development of more computationally efficient algorithms, as suggested by recent work [Jaikaumar 78, Hochbaum 78, Fisher 78] in related problem areas, would prove to be of tremendous benefit. Aside from allowing the completion of similar comparative studies on more complex data structures found in production applications, the availability of computationally efficient solution procedures allows us to develop design tools for the DBA. These tools can both assist the DBA or directly automate portions of his task.

It is desirable that the analytic models developed in this research be extended to reflect a full, high-level language implementation rather than a set-level language. Once this is accomplished, the next logical step is to compare some relational implementations [Stonebraker 77] with the high-level CODSYL implementations using compatible models. How do these alternative data models compare in performance for the same database structures and transaction mixes? Do the relational query optimization models truly improve performance when realistic, control overhead costs are assumed?

7.3 Concluding Remarks

The construction of a distributed database is a difficult task. DDBMS software will be successful if it simplifies the construction of distributed databases and does so in an efficient manner. In this study we have attempted to provide efficient, geographically transparent access to a distributed database to simplify the application programming task. Contributions to accomplish this in a homogeneous data model environment (CODASYL) were made, but work remains to be accomplished. And the challenge remains to do the same in a heterogeneous data model environment with DDBMS software, which is based on either a top-down or bottom-up design process.

Appendix

Performance Model Equation Development

1.0 Given Information for All Models

This appendix presents the details of the models used in chapter 5. All models developed use the common set of information discussed below. Some of the models require additional information which is discussed in the respective sections for those models.

1.1 Index Sets

The integer decision variables are drawn from the index sets below.

$\mathbf{R} = [1, \ldots, i, \ldots, n]$	the record index set
$\mathbf{S} = [1, \ldots, j, \ldots, m]$	the set index set
$\mathbf{L} = [1, \ldots, l, \ldots, m, m+1]$	the location modes
$\mathbf{H} = [1, \ldots, h, \ldots, w]$	the hosts
$\mathbf{U} = [1, \ldots, k, \ldots, u]$	the set implementations
$\mathbf{Q} = [1, \ldots, q, \ldots, v]$	the transaction classes

The index sets **R**, **S**, **H**, and **L** derive from the global schema: n record types, m set types, and w hosts. There are m+1 location modes: m of these are via set j and one is the CALC location mode.

The index set **U** is dependent on the DBMS implementation. In SEED, u equals 5 since sets can be implemented in 5 ways: chain with next pointers, chain with next and owner pointers, chain with next and prior pointers, chain with next, prior, and owner pointers, and pointer array.

When a variable is drawn from an index set and the index used is the same as that used to define the index set, the index set is not explicitly indicated. For example, if in a summation an i is used as the index variable, the index set **R** is implied. When multiple indices from the same index set are required, index variables with primes are used. For example, when several variables are required from the **R** index set, i, i′, and i″ can be used.

1.2 Characteristics of the Computer System

BSIZE	is the block size in bytes, 640.
$G1_h$	is the capacity of primary storage in bytes.
$G2_h$	is the capacity of secondary storage in bytes.
C1	is the cost of one-byte second of primary storage, 4.88E–7.
C2	is the cost to store 1 block on disk for 1 month, 7.5E–2.
T1	is the average time for one sequential disk access, 75ms.
T2	is the average time for one random disk access, 75ms.

1.3 Communications Functions

CT(hf,ht,1)	is the time in seconds to send a message of length 1 from host hf to host ht.
CC(hf,ht,1)	is the cost in dollars to send a message of length 1 from host hf to host ht.

1.4 DBMS Characteristics (SEED)

Q_k,	$k \in$ **U**; Q_k=1 implies that set implemented as type k has owner pointers. In SEED, Q = [0,1,0,1,1].
Q'_k,	$k \in$ U; Q'_k=1 implies that set implemented as type k has prior pointers. In SEED, Q = [0,0,1,1,0].
c,	is the size of a DBMS pointers. In SEED, c is 5 bytes.
KM_k,	$k \in$ **U**, is the number of bytes in a member record occurrence to implement set implementation K. In SEED, KM = [2, 2c, 2c, 3c, c].

KO_{jk},	$j \in \mathbf{S}$, $k \in \mathbf{U}$, is the number of bytes in an owner record occurrence of set type j with set implementation k. For SEED, KO = [c, c, 2c, 2c, c E_j].
CM,	length of database command, 20 characters.

1.5 Schema Structure

O_{ij}	$i \in \mathbf{R}$, $j \in \mathbf{S}$. Record type i owns set type j.
M_{ij}	$i \in \mathbf{R}$, $J \in \mathbf{S}$. Record type i is a member of set j.
DE_i	$i \in \mathbf{R}$. Record type i size in bytes.

1.6 Data Record Occurrence Frequency

F_{ih}	$i \in \mathbf{R}$, $h \in \mathbf{H}$. Number of record type i instances originating at host h.

1.7 Transaction Structure

AR_{gq}	number of times transaction q is run from host g.
AP_{qig}	number of accesses to all record type i (for all) to support transaction q from host g.
APR_{qig}	number of accesses to record type i (for all), restricted to a host, to support transaction q from host g.
APP_{qig}	number of accesses to record type i by key (for one) to support transaction q from host g.
A_{qij}	number of accesses to record i using set j to support transaction q.
U_{qi}	equals 1 if record type i is updated by transaction q.
AA_{qi}	equals 1 if record type i added by transaction q.

1.8 Memory Used Function

The amount of memory used at each host is a function of the number of buffers, $\mathbf{B}_h$, allocated at that host. Also, F3 = (Seed Kernel Size) + $\mathbf{B}_h$ BSIZE. The SEED kernel is approximately 30,000 bytes.

1.9 Hashing Overflow Function

The hashing overflow function simulates the additional accesses required because of hashing collisions. Overflow depends upon the amount of free space in the area. After a review of experimental results [Martin 75, page 372], for 80 percent full files, a 0.1 estimate was derived. The 0.1 indicates that record access time by key is increased 10 per cent, on the average, because of overflow records. Although the 0.1 represents a constant function, in the general case the F2 function is dependent upon $\mathbf{EP}_h$, the area fullness ratio.

$$F2(\mathbf{EP}_h) = .1$$

2.0 The RAM Model Equations

2.1 The Decision Variables for the RAM Model

$\mathbf{X}_{ij}$,	$i \in \mathbf{R}$, $j \in \mathbf{L}$, $\mathbf{X}_{ij} = 0, 1$. If $\mathbf{X}_{ij} = 1$, then the ith record has jth location mode.
$\mathbf{Y}_{jk}$,	$j \in \mathbf{S}$, $k \in \mathbf{Q}$, $\mathbf{Y}_{jk} = 0, 1$. If $\mathbf{Y}_{jk} = 1$, then the jth set has kth implementation.
$\mathbf{Z}_i$,	$i \in \mathbf{R}$, $\mathbf{Z}_i = 0, 1$. If $\mathbf{Z}_i = 1$, then the ith record is in a singular set.
$\mathbf{H}_{ih}$,	$i \in \mathbf{R}$, $h \in \mathbf{H}$, $\mathbf{H}_{ih} = 0, 1$. If $\mathbf{H}_{ih} = 1$, then the ith record is assigned to host h.
$\mathbf{B}_h$,	the number of pages for buffers at host h.
$\mathbf{EP}_h$,	the fullness ratio of the data area at host h. Fullness is 1 minus the free space ratio.

2.2 Derived Data to Support RAM Objective Function

2.2.2 Number of Records of Type I

FF_i, the number of records of type i, is the sum of type i records originating at each host:

$$FF_i = \Sigma_h F_{ih}, \text{ for all i.}$$

2.2.2 Average Number of Records in a Set Instance

E_j, the average number of records in set type j, is the number of member records divided by the number of owner records:

$$E_j = (\Sigma_i M_{ij} Fi)/(\Sigma_i O_{ij} Fi), \text{ for all j.}$$

2.2.3 Record Size with Overhead

ZE_i, the size of a data record with overhead, is the data record size plus the overhead for all stored pointers. ZE_i depends upon the set implementations (**Y**), the record location modes (**X**), and the singular set decisions (**Z**).

$$\begin{aligned} ZE_i &= f(\mathbf{Y}_{jk}, \mathbf{X}_{i,m+1}, \mathbf{Z}_i) \\ &= DE_i + c + \Sigma_j (O_{ij} \Sigma_k (KO_{jk} \mathbf{Y}_{jk})) \\ &+ \Sigma_j (M_{ij} \Sigma_k (KM_k \mathbf{Y}_{jk})) \\ &+ c \mathbf{X}_{i,m+1} \\ &+ c \mathbf{Z}_i. \end{aligned}$$

2.2.4 Number of Disk Pages at Host H

P_h, the number of pages (blocks) of disk at host h, is the total size of all records at host h, including overhead, divided by the effective block size at that host. The effective block size is the actual block size times the area fullness ratio at that host.

$$P_h = (\Sigma_i \mathbf{H}_{ih} FF_i ZE_i)/(BSIZE\ \mathbf{EP}_h)$$

2.2.5 Via Cluster Component Size

The average size of a via cluster at host h of set j containing record type i, $CLSTSZ_{ijh}$, can be calculated as follows:

$$CLSTSZ_{ijh} = \text{CEIL [byte-size-of-cluster/effective-block-size]}.$$

CEIL represents the ceiling or largest integer function. The byte-size-of-cluster is ZE_i times E_j. The effective-block-size is BSIZE times $\mathbf{EP}_h$.

2.2.6 Remote Disk Access Times/Costs

The time/cost of accessing a remote disk block at host h from host g is the local time/cost plus the respective communications time/cost. CM is the length of a data request and BSIZE is the size of the returned disk block.

$$\begin{aligned} TD_{gh} &= \text{remote disk access time} \\ &= F1(h)\ T2 + CT(g,h,CM) + CT(h,g,BSIZE) \end{aligned}$$

$$\begin{aligned} TS_{gh} &= \text{remote sequential disk access time} \\ &= T1 + CT(g,h,CM) + CT(h,g,BSIZE) \end{aligned}$$

$$\begin{aligned} CD_{gh} &= \text{remote direct access cost} \\ &= CC(g,h,CM) + CC(h,g,BSIZE) \end{aligned}$$

$$CS_{gh} = CD_{gh}$$

2.2.7 Time to Access Records in a Set Instance

PAT_{gj} is the time to access instances of a set from host g.

$$PAT_{gj} = \Sigma_i\ M_{ij}\ \Sigma_h \mathbf{H}_{ih}\ [\mathbf{X}_{ij}\ VIA_{ijgh} + (1-\mathbf{X}_{ij})\ NON\text{-}VIA_{ijgh}],$$

$$\begin{aligned} \text{where } VIA_{ijgh} &= \text{time to access members of a via set} \\ &= (CLSTSZ_{ijh} - 1)\ TS_{gh}, \\ \text{and } NON\text{-}VIA_{ijgh} &= \text{time to access non-via set members} \\ &= E_j\ TD_{gh}. \end{aligned}$$

2.2.8 Average Transaction Processing Time

$TBAR_{gq}$ represents the average transaction processing time for transaction q form host g. $TBAR_{gq}$ is composed of four components:

- TI_{gqih}, the time to access the entry records (for all);
- TII_{gqih}, the time to access a single entry record by key (for one);
- $TIII_{gqijh}$, the time to access records as members of sets; and
- TIV_{gqijh}, the time to access records as owners of sets.

$$TBAR_{gq} = \Sigma_i \Sigma_j \Sigma_h [TI_{gqih} + TII_{gqih} + TIII_{gqijh} + TIV_{gqijh}]$$

The component equations will now be developed. In all cases, access to retrieved records, i.e., not inserted records, is calculated. For this reason, the $(1-AA_{qi})$ factor is included in all component equations. The changes necessary for record addition are developed in section 2.3.

2.2.8.1 TI_{gqih} component equation. TI_{gqih} has two terms. If the record is in a singular set, then access time is the number of records times the time for a remote direct disk access. If the record is not in a singular set, then access time is based on a sequential area search, which is the number of pages at that host times the time for a remote sequential disk access.

$$TI_{gqih} = AP_{qig} [\mathbf{Z}_i\ FF_i\ TD_{gh} + (1-\mathbf{Z}_i)\ \mathbf{H}_{ih}\ P_h\ TS_{gh}]\ (1-AA_{qi})$$

2.2.8.2 TII_{gqih} component equation. TII_{gqih} has three terms. If the record does not have CALC location mode and keyed access is desired, the record must be found by performing a "for all" search as discussed in the TI_{gqih} equation development. If the record has CALC location mode and keyed access is desired, then access time is the time to perform a remote direct access taking into account the probability of performing additional overflow accesses due to CALC chain overflow.

$$\begin{aligned} TII_{gqih} = APP_{qig} [&(1-\mathbf{X}_{i,m+1})\ \mathbf{Z}_i\ FF_i\ .5\ TD_{gh} \\ &+ (1-\mathbf{X}_{i,m+1})\ (1-\mathbf{Z}_i)\ .5\ \mathbf{H}_{ih}\ P_h\ TS_{gh} \\ &+ \mathbf{X}_{i,m+1}\ (1 + F2\ (\mathbf{EP}_h))\ TD_{gh}]\ (1-AA_{qi}) \end{aligned}$$

2.2.8.3 $TIII_{gqijk}$ component equation. The time to access records in a set instance has already been defined and is represented by the PAT_{gj} calculation. Using this result, the development of $TIII_{gqijh}$ follows directly.

$$TIII_{gqijh} = A_{qij}\ AR_{qg}\ M_{ij}\ PAT_{gj}\ (1-AA_{qi})$$

2.2.8.4 TIV_{gqijh} component equation. To simplify the TIV_{gqijh} equation development, an equation for owner access time, similiar to member access time, is developed. Let $PATO_{ijgh}$ be the time to access the owner record of set j from record i from host g. $PATO_{ijgh}$ is one of three terms selected depending upon whether set j is a via set with owner pointers, a non-via set with owner pointers, or a set with no owner pointers.

$$PATO_{ijgh} = \Sigma_k \, [\mathbf{Y}_{jk} \, \mathbf{X}_{ik} \, Q_k \, BETA_{ijh} \, TS_{gh} + (1-\mathbf{X}_{ij}) \, Q_k \, TD_{gh} + (1-Q_k) \, .5 \, PAT_{gj}]$$

where $BETA_{ijh}$ = $(1- (1/CLSTSZ_{ijh}))$, a correction factor which reflects the fact that a via set owner may be on the same page as the member record thereby not requiring further disk access.

$$TIV_{gqijh} = A_{gij} \, AR_{qg} \, O_{ij} \, PATO_{ijgh} \, (1-AA_{qi})$$

2.2.9 Cost To Access Records in a Set Instance

PAC_{gj} is the cost to access instances of set j from host g. PAC_{gj} equation development is analogous to PAT_{gj} equation development.

$$PAC_{gj} = \Sigma_i \, M_{ij} \, \Sigma_h \, \mathbf{H}_{ih} \, [\mathbf{X}_{ij} \, VIAC_{ijgh} + (1-\mathbf{X}_{ij}) \, NON\text{-}VIAC_{ijgh}],$$

where $VIAC_{ijgh}$ = cost to access members of a via set
= $(CLSTSZ_{ijh} - 1) \, CS_{gh}$

and $NON\text{-}VIAC_{ijgh}$ = cost to access non-VIAC set members
= $E_j \, CD_{gh}$.

2.2.10 Average Transaction Processing Cost

$CBAR_{gq}$ represents the average transaction processing cost for transaction q from host g. $CBAR_{gq}$ is composed of four components:

- CI_{gqih}, the cost to access the entry records (for all);
- CII_{gqih}, the cost to access a single entry record by key (for one);
- $CIII_{gqijh}$, the cost to access records as members of sets; and
- CIV_{gqijh}, the cost to access records as owners of sets.

$$CBAR_{gq} = \Sigma_i \, \Sigma_j \, \Sigma_h \, [CI_{gqih} + CII_{gqih} + CIII_{gqijh} + CIV_{gqijh}].$$

$$CI_{gqih} = AP_{qig} \, [\mathbf{Z}_i \, FF_i \, CD_{gh} + (1-\mathbf{Z}_i) \, \mathbf{H}_{ih} \, P_h \, CS_{gh}] \, (1-AA_{qi}).$$

$$CII_{gqih} = APP_{qig} \, [(1-\mathbf{X}_{i,m+1}) \, \mathbf{Z}_i \, .5 \, CD_{gh} + (1-\mathbf{X}_{i,m+1}) \, (1-\mathbf{Z}_i) \, .5 \, \mathbf{H}_{ih} \, P_h h \, CS_{gh} + \mathbf{X}_{i,m+1} \, (1 + F2(\mathbf{EP}_h)) \, CD_{gh}] \, (1-AA_{qi}).$$

$$CIII_{gqijh} = A_{qij}\ AR_{qg}\ M_{ij}\ PAC_{gj}\ (1-AA_{qi}).$$

$$CIV_{gqijh} = A_{qij}\ AR_{qg}\ O_{ij}\ PACO_{igjh}\ (1-AA_{qi}),$$

where $PATO_{ijgh}$ is the cost to access the owner record of set j from record i from host g.

$$PACOigjh = \Sigma_k\ [Y_{jk}\ \mathbf{X}_{ik}\ Q_k\ BETAijh\ CSgh + (1-\mathbf{X}_{ij})\ Q_k\ CD_{gh} + (1-Q_k)\ .5\ PAC_{gj}],$$

where $BETA_{ijh} = (1-(1/CLSTSZ_{ijh}))$, a correction factor which reflects the fact that a via set owner may be on the same page as the member record thereby not requiring further disk access.

2.3 Modifications to Reflect Update Costs

When records are added or updated, both additional communications costs and disk access costs are incurred. Additional communications time and cost are incurred to transmit the updated disk block to its location site. If the record is stored where it originates, these charges are not incurred. Additional disk access time is required to write or rewrite the disk block for the new or updated record.

However, a major expense occurs for added records which are set members. the owner records for each of the sets which the added record is in must be retrieved and rewritten to reflect the addition of the new record. If the owners are remote from the member record, additional communications costs must also be incurred.

2.3.1 Updating Non-Keyed Records

The additional time to add/update a non-keyed record, $UATI_{gqih}$, is the time to write/rewrite the disk block plus the time to link an added record into a singular set.

$$UATI_{gqih} = TD_{gh} + \mathbf{Z}_i\ TD_{gh}\ AA_{qi}$$

The additional communications cost, $UACI_{gqih}$, is defined in a similar fashion.

$$UACI_{gqih} = CD_{gh} + \mathbf{Z}_i\ CD_{gh}\ AA_{qi}$$

The "for all" time/cost equations are updated to include this additional factor:

$$TI'_{gqih} = TI_{gqih} + APP_{qig}\ UATI_{gqih}\ (AA_{qi} + U_{qi})$$
$$CI'_{gqih} = CI_{gqih} + APP_{qig}\ UACI_{gqih}\ (AA_{qi} + U_{qi})$$

2.3.2 Updating Keyed Records

The additional time to add/update a keyed record, $UATII_{gqih}$, is the time to rewrite the updated record or the time to add a new record to a singular set plus the time to add the record using calc location mode.

$$\begin{aligned} UATII_{gqih} &= TD_{gh} + \mathbf{Z}_i\ TD_{gh}\ AA_{qi} \\ &\quad + AA_{qi}\ TD_{gh}\ (1-\mathbf{X}_{i,m+1})\ (1+ F2(h)) \end{aligned}$$

The additional communications cost, $UACII_{gqih}$, is similiarly defined.

$$\begin{aligned} UACII_{gqih} &= CD_{gh} + \mathbf{Z}_i\ CD_{gh}\ AA_{qi} \\ &\quad + AA_{qi}\ CD_{gh}\ (1-\mathbf{X}_{i,m+1})\ (1+ F2(h)) \end{aligned}$$

The "for one by key" time/cost equations are updated to include this additional factor:

$$TII'_{gqih} = TII_{gqih} + APP_{qig}\ UATII_{gqih}\ (AA_{qi} + U_{qi}).$$
$$CII'_{gqih} = CII_{gqih} + APP_{qig}\ UACII_{gqih}\ (AA_{qi} + U_{qi}).$$

2.3.3 Updating Member Records

The time to add/update a member record includes the record maintenance plus the set linkage maintenance. The record maintenance is similiar to that derived above for keyed records, except it is now possible to add a record to a via set. The record maintenance time equation has four components: the time to rewrite the disk block containing the updated record; the time to add the record to a singular set; the time to insert the record into a via set; and the time to insert the record using a calc key. Insertion of the record into any non-via sets is included in the set linkage maintenance costs derived below.

$$\begin{aligned} UATIII_{gqih} &= TD_{gh}\ U_{qi} + \mathbf{Z}_i\ TD_{gh}\ AA_{qi} \\ &\quad + AA_{qi}\ \mathbf{X}_{ij}\ TS_{gh}\ ZE_i/(BSIZE\ \mathbf{EP}_h) \\ &\quad + AA_{qi}\ (1-\mathbf{X}_{i,m+1})\ (1 + F2(h) + 1)\ TD_{gh} \end{aligned}$$

$$\begin{aligned} UACIII_{gqih} = {} & CS_{gh}\ U_{qi} + \mathbf{Z}_i\ CS_{gh}\ AA_{qi} \\ & + AA_{qi}\ \mathbf{X}_{ij}\ CS_{gh}\ ZE_i/(BSIZE\ \mathbf{EP}_h) \\ & + AA_{qi}\ (1-\mathbf{X}_{i,m+1})\ (1 + F2(h) + 1)\ CD_{gh} \end{aligned}$$

To calculate set update time, UPDSET(h′,h″,i,j,g), and set update cost, UPDSEC(h′,h″,i,j,g), the owner and member location must be determined along with set implementation mode for all sets that the record is in. Expressions for these quantities will be derived. The time to update a set linkage has two components: the time to get and rewrite the owner record and the time to get and rewrite the prior record. The cost to update a set linkage has similiar components.

$$\begin{aligned} UPDSET(h',h'',i,j',g) = {} & PATOigj'h'' + TD_{gh''} \\ & + PATP_{igj'h''} + TD_{gh''} \end{aligned}$$

$$\begin{aligned} UPDSEC(h',h'',i,j',k) = {} & PACOigj'h'' + CD_{gh} \\ & + PACP_{igj'h''} + CD_{gh''} \end{aligned}$$

where $PATP_{igj'h''}$ is the time to find the prior record

$$PATP_{igj'h''} = \Sigma_k\ Y_{jk}\ Q'_k\ TD_{gh''} + \Sigma_k\ Y_{jk}\ (1-Q'k)\ .5\ PAT_{gj}$$

where $PACP_{igj'h''}$ is the cost to find the prior record

$$PACP_{igj'h''} = \Sigma_k\ Y_{jk}\ Q'_k\ TD_{gh''} + \Sigma_k\ Y_{jk}\ (1-Q'k)\ .5\ PAC_{gj}$$

$$\begin{aligned} & UATIII_{gqijh} = UATIII_{gqijh} + \\ & \Sigma_{j'\neq j}\ M_{ij'}\ \Sigma_{i'}\ O_{i'j'}\ \Sigma_{h''}\ \mathbf{H}i',h''\ UPDSET(h',h'',i,j',g) \end{aligned}$$

$$\begin{aligned} & UACIII_{gqijh} = UACIII_{gqijh} + \\ & \Sigma_{j'\neq j}\ M_{ij'}\ \Sigma_{i'}\ O_{i'j'}\ \Sigma_{h''}\ \mathbf{H}i',h''\ UPDSEC(h',h'',i,j',g) \end{aligned}$$

The member time/cost equations are modified:

$$TIII'_{gqijh} = TIII_{gqijh} + A_{qij}\ AR_{gq}\ (AA_{qi} + U_{qi}\ UATIII_{gqijh}$$

$$CIII'_{gqijh} = CIII_{gqijh} + A_{qij}\ AR_{gq}\ (AA_{qi} + U_{qi}\ UACIII_{gqijh}$$

To update an owner record just requires a remote record rewrite. A record cannot be added through "to owner" access.

$$\begin{aligned} UATIV_{gqijh} &= TD_{gh} \\ UACIV_{gqijh} &= CD_{gh} \end{aligned}$$

The time/cost equations are updated:

$$TIV'_{gqijh} = TIV_{gqijh} + A_{qij}\ AR_{gq}\ U_{qi}\ UATIV_{gqijh}$$

$$CIV'_{gqijh} = CIV_{gqijh} + A_{qij}\ AR_{gq}\ U_{qi}\ UACIV_{gqijh}$$

2.3.4 Revised TBAR/CBAR Equations

The equations for average transaction processing time, $TBAR_{gq}$, and average transaction processing cost, $CBAR_{gq}$, are changed to reflect update by using the component equations developed in the update section. These are the equations with the primed (′) terms.

2.4 RAM Objective Function

The RAM objective function is to minimize costs:

min {comm. costs + access-costs + disk costs}
$\min \{\Sigma_q\ \Sigma_g\ [C1\ F3\ TBAR_{gq} + CBAR_{gq}] + \Sigma_h\ P_h\ C2\}$.

2.5 RAM Model Constraints

The following constraints are applicable to the RAM model.

1. $\Sigma_j\ \mathbf{X}_{ij} = 1,\quad i=1,2,\ldots m$
   (one location mode per record)

2. $\mathbf{X}_{ij} = M_{ij},\quad i=1,2,\ldots n,\quad j=1,2,\ldots,m$
   (via a set which it is in)

3. $\Sigma_k\ \mathbf{Y}_{jk} = 1,\quad j=1,2,\ldots,m$
   (one set implementation per set)

4. $P_h\ BSIZE \leq G1_h,$ for all h
   (disk constraint at each host)

5. $\mathbf{B}_h\ BSIZE \leq G2_h,$ for all h
   (memory constraint at each host)

6. $\Sigma_h\ \mathbf{H}_{ih} = 1,$ for all h
   [$>$ 1 for data redundancy]
   (each record type at one location only)

7. $\mathbf{H}_{kh}\ [\Sigma_k\ O_{kj}\ \mathbf{H}_{kh}] >= \mathbf{X}_{ij}$, for all i, j, h
   (via set owner and members at same host)

8. $TBAR_{gq} \leq RR_{gq}$, for all q, g
   (turn-around time constraint)

3.0 The RDH Model Equations

3.1 The Decision Variables for the RDH Model

The RDH model has the same decision variable set as the RAM model, plus an additional decision, the record location-mode binding time decision, **R**.

$$\mathbf{R}_i\ i \in \mathbf{R}, \quad \mathbf{R}_i = 0,\ 1.$$

If $\mathbf{R}_i = 1$, then record i has static location mode, otherwise the record i has dynamic-local location mode. Dynamic-mobile location mode is not addressed by the model.

3.2 Derived Data To Support The RDH Objective Function

3.2.1 Average Number of Records in Set J at a Given Host

When a set is distributed, member records of that set can exist at multiple hosts different from the owner record's location. That collection of member records is referred to as the *distributed set component*. The number of records in the distributed set component, E_{jh}, can be determined if the origin of location is used to define the membership of the set components. A set can be distributed if the member record has dynamic-local location mode. It can also be distributed if both owner and member have static location mode and are located at different locations. However, this case does not actually have a distributed set. All the members are located at one location and the number of members in a set at that location is the same as E_j for the set.

If the owner record type has static location mode:

$$E_{jh} = (\Sigma_i\ M_{ij}\ F_h)/(\Sigma_i\ O_{ij}\ FF_i), \quad \text{for all j, h.}$$

If the owner record type has dynamic-local location mode:

$$E_{jh} = (\Sigma_i\ M_{ij}\ F_{ih})/(\Sigma_h\ \Sigma_i\ O_{ij}\ FF_{ih}), \quad \text{for all j, h.}$$

3.2.2 Record Size with Overhead

ZE_i, the size of a data record with overhead, is the data record size plus the overhead for all stored pointers.

$$\begin{aligned} ZE_i = {} & DE_i + c \\ & + \Sigma_j \left(O_{ij}\ \Sigma_k\ KO_{jk}\ \mathbf{Y}_{jk}\right) \\ & + \Sigma_j \left(M_{ij}\ \Sigma_k\ KM_k\ \mathbf{Y}_{jk}\right) \\ & + c\ \mathbf{X}_{i,m+1} \\ & + c\ \mathbf{Z}_i \end{aligned}$$

3.2.3 Number of Disk Pages at Host H

P_h, the number of disk pages at host h, is the total size of all records at host h including overhead converted to blocks.

$$P_h = \Sigma_i\ [\mathbf{R}_i\ \mathbf{H}_{ih}\ FF_i\ ZE_i + (1-\mathbf{R}_i)\ F_{ih}\ ZE_i]/(BSIZE\ \mathbf{EP}_h)$$

3.2.4 Data Record Request/Return Time and Cost

Let CTF_{gih}/CCF_{gih} be the command transmission and associated return data transmission time/cost.

$$CTF_{gih} = CT(g,h,CM) + CT(h,g,De_i)$$

$$CCF_{gih} = CC(g,h,CM) + CC(h,g,De_i)$$

3.2.5 Disk Access Times

Let TD_h be the direct access time at host h. Let TS_h be the sequential access time at host h.

$$TD_h = T2\ F1(h) \quad TS_h = T1$$

3.2.6 Set Access Time

Define $DPAT_{gqijhi'}$ to be the time to access all members of a set j to support transaction q from host g, where h′ and h represent the owner record, identified by i′, and the member record, identified by i, locations, respectively, of the set j.

$$\begin{aligned} DPAT_{gqijhi'} &= \text{(time to send requests from g to member site, h)} \\ &+ \text{(time to access member records)} \\ &+ \text{(time to send member records from their site, h, to g).} \end{aligned}$$

Member record access time will be further increased if the set is distributed. Both the distributed and non-distributed set cases can have member records which are in via or non-via sets. In the distributed via case, the member records are stored locally about a local set header record.

The $DPAT_{gqijhi'}$ equation is developed in section 3.3.3 using the distributed and non-distributed set processing formulas defined below.

3.2.6.1 Non-distributed set access time. Let $PATN_{gqijh}$ be the access time for member records i of non-distributed set j.

$$PATN_{gqijh} = E_j\,[CT(g,h,CM) + CT(h,g,DE_i)] + DAT_{gqijh},$$

where DAT_{gqijh} is the disk access time for the member records.

$$DAT_{gqijh} = \mathbf{X}_{ij}\,(CLSTSZ_{ijh} - 1)\,TS_h + (1-\mathbf{X}_{ij})\,E_j\,TD_h,$$

where $CLSTSZ_{ijh}$ = the via cluster size

$$= CEIL\,[ZE_i\,E_j)/(BSIZE\,\mathbf{EP}_h)].$$

3.2.6.2 Distributed set component access time. Let $PATD_{gqijh}$ be the access time for member records i of distributed set j.

$$PATD_{gqijh} = \Sigma_h\,E_{jh}\,[CT(g,h,CM) + CT(h,g,DE_i)] + DDAT_{gqijh},$$

where $DDAT_{gqijh}$ is the disk access time for the set component at h.

$$DDAT_{gqijh} = [\mathbf{X}_{ij}\,(CLSTSZ_{ijh} - 1)\,TS_h + (1-\mathbf{X}_{ij})\,E_{jh}\,TD_h],$$

where $CLSTSZ_{ijh}$ is the size of the local member cluster.

$$CLSTSZ_{ijh} = CEIL\,[(ZE_i\,E_{jh})/(BSIZE\,\mathbf{EP}_h)]$$

As in the integrated data base model [Gerritsen 77a], the transitive nature of via clustering is not explicitly modeled. The via location of distributed set member records results in the formation of clusters about a local set header. Access to owner host site pointers is assumed to be free, i.e., when the owner record is accessed, this small set of host pointers, located via, is also accessed.

3.2.7 Set Access Cost

Define $DPAC_{gqijhi'}$ to be the cost to access all members of a set j to support transaction q from host g, where h′ and h represent the owner and member record locations, respectively, of the set j.

$$DPAC_{gqijhi'} = \text{(cost to send requests from g to member site, h)} + \text{(cost to send member records from their site, h, to g).}$$

Member record access cost will be further increased if the set is distributed. Both the distributed and non-distributed set cases can have member records which are in via or non-via sets. In the distributed via case, the member records are stored locally about a local set header record.

3.2.7.1 Non-distributed set access cost. Let $COMN_{gqijh}$ be the access cost for member records i of non-distributed set j.

$$COMN_{gqijh} = E_j\ [CT(g,h,CM) + CT(h,g,DE_i)]$$

3.2.7.2 Distributed set component access cost. Let $COMD_{gqijh}$ be the access cost for member records i of distributed set j.

$$COMD_{gqijh} = \Sigma_h\ E_{jh}\ [CT(g,h,CM) + CT(h,g,DE_i)]$$

3.3 Average Transaction Time

$TBAR_{gq}$, the average transaction time to process a transaction q from host g, has four components: TI_{gq}, the time to access the entry records (for all; TII_{gq}, the time to access a single entry-record (for one); $TIII_{gq}$, the time to access records as members of sets; and TIV_{gq}, the time to access records as owners of sets.

$$TBAR_{gq} = TI_{gq} + TII_{gq} + TIII_{gq} + TIV_{gq}$$

$CBAR_{gq}$, the average transaction cost for transaction q run from host g is similiarly defined:

$$CBAR_{gq} = CI_{gq} + CII_{gq} + CIII_{gq} + CIV_{gq}$$

3.3.1 "For All" Access

The equation development for TI_{gq} has both static and dynamic-local location mode terms.

$$\begin{aligned} TI_{gq} &= \Sigma_i\ AP_{qig}\ \{(\text{static}) + (\text{dynamic-local})\} \\ &\{(\mathbf{R}_i)\ \Sigma_h\ \mathbf{H}_{ih}\ [CTF_{gih}\ FF_i + \mathbf{Z}_i\ FF_i\ TD_h + (1-\mathbf{Z}_i)\ P_h\ TS_h] \\ &+ (1-\mathbf{R}_i)\ \Sigma_h\ [CTF_{gih}\ F_{ih} + \mathbf{Z}_i\ F_{ih}\ TD_h + (1-\mathbf{Z}_i)\ P_h\ TS_h]\} \end{aligned}$$

The equation for CI_{gq} is similar.

$$\begin{aligned} CI_{gq} &= \Sigma_i\ AP_{qig}\ \{(\text{static}) + (\text{dynamic-local})\} \\ &\{\Sigma_h\ \mathbf{H}_{ih}\ [(\mathbf{R}_i\ CCF_{gih}\ FF_i + (1-\mathbf{R}_i)\ CCD_{gih}\ F_{ih}]\} \end{aligned}$$

3.3.2 "For One"Access

TII_{gh} has both static and dynamic-local terms.

$$\begin{aligned} TII_{gq} &= \Sigma_i\ APP_{qig}\ \{(\text{static}) + (\text{dynamic-local})\} \\ &\{\Sigma_h\ [\mathbf{R}_i\ \mathbf{H}_{ih}\ TAH_{gih} + (1-\mathbf{R}_i)\ TAH_{gih}/2]\}, \end{aligned}$$

where TAH_{gih} is the time to ask host h from host g if record i exists.

Assuming no parallel polling, on the average half of the hosts will have to be polled. If the record exists, it is returned.

$$\begin{aligned} TAH_{gih} &= CT(g,h,CM) + CT(h,g,DE_i) \\ &+ (1-\mathbf{X}_{i,m+1})\ \mathbf{Z}_i\ (.5\ FF_i)\ TD_h \\ &+ (1-\mathbf{X}_{i,m+1})\ (1-\mathbf{Z}_i)\ (.5\ \mathbf{P}_h)\ TS_h \\ &+ \mathbf{X}_{i,m+1}\ (1 + F2\ (\mathbf{EP}_h))\ TD_h \end{aligned}$$

The equation for CII_{gq} is similiarly defined.

$$\begin{aligned} CII_{gq} &= \Sigma_i\ APP_{qig}\ \{(\text{static}) + (\text{dynamic-local})\} \\ &\Sigma_i\ APP_{qig}\ \{\Sigma_h\ [\mathbf{R}_i\ CAH_{gih} + (1-\mathbf{R}_i)\ CAH_{gih}/2]\} \end{aligned}$$

where CAH_{gih} is the cost to ask host h if the record exists, $CAH_{gih} = CC(g,h,CM) + CC(h,g,DE_i)$

3.3.3 "To Member" Access

In order to develop the equations for access to member records, an exhaustive enumeration of situations was done. Since a member record can have both static and dynamic-local location mode and the owner of the associated set can have static and dynamic-local location mode, four cases are possible. However, in some of these cases, the fact that the set is distributed can give rise to additional cases. Three relevant cases result after sorting everything out: the record has static location mode; the record has dynamic-local location mode and the owner of the set has static location mode; and the record has dynamic-local location mode and the owner of the set has dynamic-local.

The equation for access to a set is given below:

$$\begin{aligned} DPAT_{gqijhi'} = {} & (\mathbf{R}_i)\ PATN_{gqijh} \\ & + (1-\mathbf{R}_i)\ (\mathbf{R}_{i'})\ PATD_{gqijh} \\ & + (1-\mathbf{R}_i\ (1-\mathbf{R}_{i'})\ PATD_{gqijh}/NP_j \end{aligned}$$

where i is the member record type,
i′ is the owner record type of set j,
and NP_j is the number of possible sites with distributed set components.

The NP_j calculation can be represented algebraically as:

$$NP_j = \Sigma_{h''}\ (E_{jh''}/E_{jh''}).$$

The communications cost to access a set, $DCOM_{gqijhi'}$, is similiarly defined:

$$\begin{aligned} DCOM_{gqijhi'} = {} & (\mathbf{R}_i)\ COMN_{gqijh} \\ & + (1-\mathbf{R}_i)\ (\mathbf{R}_{i'})\ COMD_{gqijh} \\ & + (1-\mathbf{R}_i)\ (1-\mathbf{R}_{i'})\ COMD_{gqijh}/NP_j. \end{aligned}$$

3.3.4 "To Owner" Access

Let PAO_{gqj} be the time to find and get the owner of set j.

$$\begin{aligned} PAO_{gqj} = {} & \Sigma_i\ M_{ij}\ \Sigma_{i'}\ O_{ij}\ \Sigma_k\ \mathbf{Y}_{jk}\ \Sigma_h\ \mathbf{H}_{ih} \\ & [CT(g,h',CM) + CT(h',g,DE_i) + ODSK_{ijk}], \end{aligned}$$

where $ODSK_{ijk}$ = "to owner" disk access time

$$\begin{aligned} &= [\mathbf{X}_{ij}\ Q_k\ CLSTRF_{ijh}\ TS_h + (1-\mathbf{X}_{ij})\ Q_k\ TD_h \\ &\quad + \mathbf{X}_{ij}\ (1-Q_k)\ .5\ DPAT_{gqijhi'} \\ &\quad + (1-\mathbf{X}_{ij})\ (1-Q_k)\ .5\ DPAT_{gqijhi'}], \end{aligned}$$

where $CLSTRF_{ijh} = (1 - (1/CLSTSZ_{ijh}))$,

where $DPAT_{gqijhi'}$ = time to access members of set j, derived above.

where h' = location of owner = OWNLOC(i,j,g)

where OWNLOC(i,j,g) =
 if $\mathbf{R}_i=1$ then $h' = h''$ such that $\mathbf{H}_{ih''}=1$,
 if $\mathbf{R}_i=0$ and Dj=0 then h'=g (implies no cost),
 if $\mathbf{R}_i=0$ and Dj=1 then h'=w+1 (assume a cost).

PCO_{gqj}, the cost of finding and getting an owner record, is:

$$PCO_{gqj} = \Sigma_i\ O_{ij}\ [CC(g,h',CM) + CC(h',g,DE_i)],$$

where h' = OWNLOC(i,j,g).

These equations reflect the time and cost to return the owner record. When we derive the equations for updating set membership below, only the find time will be required. Define $PATO_{gqji}$ and $PATC_{gqji}$ as the time and cost to just find the owner record.

$$PATO_{gqji} = PAO_{gqj} - CT(h',g,DE_i)$$
$$PATC_{gqji} = PCO_{gqi} - CC(h',g,DE_i)$$

where h' = OWNLOC(i,j,g)

3.3.5 "Prior" Access

The time and cost of accessing a prior record instance is required to develop equations for updating set linkages. Let $PATP_{gqijkhi'}$ and $PACP_{gqijkhi'}$ be time and cost of accessing a prior record instance for set j from host g. Let $PDSK_{gqijkhi'}$ be the local disk time to access a prior pointer.

$$\begin{aligned} PDSK_{gqijkhi'} &= [\mathbf{X}_{ij}\ Q'_k\ CLSTRF_{ijh}\ TS_h + (1-\mathbf{X}_{ij})\ Q'_k\ TD_h \\ &\quad + \mathbf{X}_{ij}\ (1-Q'_k)\ .5\ DPAT_{gqijhi'} \\ &\quad + (1-\mathbf{X}_{ij})\ (1-Q'_k)\ .5\ DPAT_{gqijhi'}] \end{aligned}$$

For both distributed and non-distributed cases:

$$PATP_{gqijkhi'} = PDSK_{gqijkhi'}, \quad PACP_{gqijkhi'} = 0.$$

Distributed prior access has no communications component because the required commands are assumed to be sent with the "to owner" commands; consequently, the costs are already included.

3.3.6 Modifications to Reflect Update

Let XDR_{qi} represent the amount of information needed for an add or update.

$$XDR_{qi} = DE_i \ AA_{qi} + DE_i \ U_{qi}$$

The time and cost to transmit this additional information is $DXMIT_{gqih}$ and $CXMIT_{gqih}$:

$$DXMIT_{gqih} = CT(g,h,XDR_{qi}) \text{ and } CXMIT_{gqih} = CC(g,h,XDR_{qi}).$$

3.3.6.1 Updating for all. Currently, the RDH model does not support updating for all. However, non-keyed records can be updated one at a time. In this case, the update time is the time to rewrite the record plus the transmission time.

$$UATI_{gqih} = TD_h + DXMIT_{gqih}$$

$$UACI_{gqih} = CXMIT_{gqih}$$

$$TI'_{gqih} = APPgqih \ UATI_{gqih} \ (AA_{qi} + U_{qi})$$

$$CI'_{gqih} = APPgqih \ UACI_{gqih} \ (AA_{qi} + U_{qi})$$

3.3.6.2 Updating for one. The time to update the record is the rewrite time plus the transmission time. If the record is added and a member of a singular set, an additional rewrite time is required.

$$UATII_{gqih} = TD_h + \mathbf{Z}_i \ TD_h \ AA_{qi} + DXMIT_{gqih}$$

$$UACII_{gqih} = CXMIT_{gqih}$$

$$\text{TII}'_{gqih} = \text{APPgqih UATII}_{gqih}\ (\text{AAgi} + \text{U}_{qi})$$

$$\text{CII}'_{gqih} = \text{APPgqih UACII}_{gqih}\ (\text{AAgi} + \text{U}_{qi})$$

3.3.6.3 Updating to members. The DAT and DDAT equations are modified.

$$\begin{aligned}\text{DAT}_{gqijh} = {} & \text{old-DAT}_{gqijh}\ (1-\text{AA}_{qi}) + \text{U}_{qi}\ \text{TD}_h \\ & + \text{AA}_{qi}\ \text{TS}_h\ \mathbf{X}_{ij}\ \text{ZE}_i/(\mathbf{EP}_h\ \text{BSIZE}) \\ & + \text{AA}_{qi}\ \text{TD}_h\ (1-\mathbf{X}_{i,m+1})\ (1 + \text{F2}(h))\end{aligned}$$

DDAT_{gqijh} is modified in a similar fashion.

$$\begin{aligned}\text{UATIII}_{gqijh} = {} & \text{TD}_h + \mathbf{Z}_i\ \text{TD}_h\ \text{AA}_{qi} + \text{DXMIT}_{gqih} \\ & + \text{UPDSET}_{gqijh}\ \text{AA}_{qi}\end{aligned}$$

$$\text{UACIII}_{gqijh} = \text{CXMIT}_{gqih} + \text{UPDSEC}_{gqijh}\ \text{AA}_{qi}$$

The time and cost of inserting a record into its member sets:

$$\begin{aligned}\text{UPDSET}_{gqijh} = {} & \Sigma_{j'\neq j}\ \text{M}_{ij}\ \Sigma_k\ \mathbf{Y}_{jk}\ \Sigma_{i'}\ \text{O}_{i'j'} \\ & [\text{PATO}_{gqj'i} + \text{TD}_h + \text{PATP}_{gqijkhi'} + \text{TD}_h]\end{aligned}$$

$$\begin{aligned}\text{UPDSEC}_{gqijh} = {} & \Sigma_{j'\neq j}\ \text{M}_{ij}\ \Sigma_k\ \mathbf{Y}_{jk}\ \Sigma_{i'}\ \text{O}_{i'j'} \\ & [\text{PACO}_{gqj'i} + \text{TD}_h + \text{PACP}_{gqijkhi'} + \text{TD}_h]\end{aligned}$$

$$\text{TIII}'_{gqijh} = \text{A}_{qij}\ \text{AR}_{gq}\ \text{UATIII}_{gqijh}\ (\text{AA}_{qi} + \text{U}_{qi})$$

$$\text{CIII}'_{gqijh} = \text{A}_{qij}\ \text{AR}_{gq}\ \text{UACIII}_{gqijh}\ (\text{AA}_{qi} + \text{U}_{qi})$$

3.3.6.4 Updating to owners.

$$\begin{aligned}\text{UATIV}_{gqijh} &= \text{TD}_h + \text{DXMIT}_{giq} \\ \text{UACIV}_{gqijh} &= \text{CXMIT}_{giq}\end{aligned}$$

$$\begin{aligned}\text{TIV}'_{gqijh} &= \text{A}_{qij}\ \text{AR}_{gq}\ (\text{UATIV}_{gqijh}\ \text{U}_{qi}) \\ \text{CIV}'_{gqijh} &= \text{A}_{qij}\ \text{AR}_{gq}\ (\text{UATIV}_{gqijh}\ \text{U}_{qi})\end{aligned}$$

3.3.7 $TBAR_{gq}/CBAR_{gq}$ Equation Modifications

In order to modify the average transaction time equation, $TBAR_{gq}$, and the average transaction cost equation, $CBAR_{gq}$, for update, the changes developed above, identified by the primed (′) equations, are inserted into the respective $TBAR_{gq}$ and $CBAR_{gq}$ equations developed earlier.

3.4 The RDH Objective Function

The RDH objective function is the same as the one used in the RAM model:

$$\min \{\text{comm. costs} + \text{access-costs} + \text{disk costs}\}$$
$$\min \{\Sigma_q \ \Sigma_g \ [C1 \ F2 \ TBAR_{gq} + CBAR_{gq}] + \Sigma_h \ \mathbf{P}_h \ C2\}.$$

3.5 RDH Constraints

The RDH constraints are the same as the RAM constraints except the via location constraint is no longer applicable. This constraint, number seven, indicated that the owner and member record of a set must be collocated.

4.0 The High-Level Data Language Model

The high-level data language is essentially a set level language. The model structure of the HLL model is similiar to the RDH model already developed. The major difference is in the way sets of records are accessed. In the RDH model, access to each record had to be individually requested. In the HLL model, only access to the set of records had to be requested with a single set of records returned. Consider access to a set of line records on an order. If the order had 10 lines, 10 requests for line records and 10 associated return data transmissions must take place for the RDH model. For the HLL model, only 2 messages must be exchanged.

Because of the similiarity of the HLL model and the RDH model, only the specific equations which have changed are developed. Aside from these changes, the HLL model is the same as the RDH model.

4.1 XDR Changes

The equation for XDR_{qi}, the amount of information needed for an add or update, is changed to reflect the transmission of only modified fields.

$$XDR_{qi} = DE_i \ AA_{qi} + DE_i \ RRQ_{qi} \ U_{qi}$$

4.2 Non-Distributed Set Processing Time

$PATN_{gqijh}$, the processing time for the member records of non-distributed set j, is changed to reflect transmission of an information chunk built from a set of records. Let $XMLN_{qij}$ be the size of the information chunk.

$$XMLN_{qij} = ZE_i \; EE_j \; RRQqi \; PRED_{qi}$$

The equation for $PATN_{gqijh}$ is then modified:

$$PATN_{gqijh} = CT(g,h,CM+XDR_{qi}) + (1-AA_{qi}) \, CT(h,g,XMLN_{qij}) + DAT_{gqijh}.$$

4.3 Distributed Set Component Processing Time Changes

$PATD_{gqijh}$, the processing time for member records of distributed set j, is changed to reflect the transmission of a data chunk. Let $XMLD_{qijh}$ be the size of the information chunk.

$$XMLD_{qijh} = ZE_i \; E_{jh} \; RRQqi \; PRED_{qi}$$

The equation for $PATD_{gqijh}$ is then modified:

$$\begin{aligned} PATD_{gqijh} = {} & \Sigma_h \; CT(g,h,CM+XDR_{qi}) \\ & + (1-AA_{qi}) \; CT(h,g,XMLD_{qijh}) + DDAT_{gqjh} \end{aligned}$$

4.4 "For All" Access Changes

Let $XMLO_{qih}$ be the size of the returned data from all instances at one host of a record type, which is distributed, i.e., instances of the type exist at multiple hosts; and let $XMLA_{qi}$ be the size of the returned data from all instances of a record type, w.

$$XMLA_{qi} = DE_i \; RRQqi \; PRED_{qi} \; FF_i$$

$$XMLO_{qih} = De_i \; RRQqi \; PRED_{qi} \; F_{ih}$$

The time and cost to transmit this information is now defined.

$$CTA_{ghqi} = CT(g,h,CM) + CT(h,g,XMLA_{qi})$$

$$CCA_{ghqi} = CC(g,h,CM) + CC(h,g,XMLA_{qi})$$

$$CTO_{ghqi} = CT(g,h,CM) + CT(h,g,XMLO_{qih})$$

$$CCO_{ghqi} = CC(g,h,CM) + CC(h,g,XMLO_{qih})$$

The equation development for TI has both static and dynamic-local location mode terms.

$$TI_{gq} = \Sigma_i\{AP_{qig}\ [\mathbf{R}_i(static) + (1-\mathbf{R}_i)\ (d-allh)] + (1-\mathbf{R}_i)\ (d-oneh)\}$$

$$static = \Sigma_h\ \mathbf{H}_{ih}\ [CTA_{ghqi}\ FF_i + \mathbf{Z}_i\ FF_i\ TD_h + (1-\mathbf{Z}_i)\ P_h\ TS_h]$$

$$d-allh = \Sigma_h\ CTO_{ghqi}\ F_{ih} + \mathbf{Z}_i\ F_{ih}\ TD_h + (1-\mathbf{Z}_i)\ P_h\ TS_h$$

$$d-oneh = \Sigma_h\ APR_{qih}\ [CTO_{ghqi}]$$

The equation for CI_{gq} is similiar.

$$CI_{gq} = \Sigma_i\{AP_{qig}\ [\mathbf{R}_i(static) + (1-\mathbf{R}_i)\ (d-allh)] + (1-\mathbf{R}_i)\ (d-oneh)\}$$

$$static = \Sigma_h\ \mathbf{H}_{ih}\ [CCA_{ghqi}]$$

$$d-allh = \Sigma_h\ AP_{qig}\ [CCO_{ghqi}]$$

$$d-oneh = \Sigma_h\ APHqih\ [CCO_{ghqi}]$$

4.5 "For One" Access Changes

$$TAH_{gih} = CT(g,h,CM) + CT(h,g,XDRgi) + \ldots$$

$$CAH_{gih} = CC(g,h,CM) + CC(h,g,XDRgi)$$

4.6 "To Member" and "To Owner" Access Changes

Changes in the low-level equations, described above, are sufficient to support "to owner" and "to member" access in the high-level language model.

5.0 The High-Level Parallel Polling Model

The high-level parallel polling model is the same as the previous high-level language model, except that "for one" record access is polled in parallel, rather then polled sequentially. Note that the same number of messages are sent, so that the communication costs are the same. The time to poll for a record is the worst case time to send the command plus the return data transmission time. Because the record can exist at any host, the return time is taken to be the average of possible return times. The TII dynamic-local term, d–allh, is changed as follows:

$$\text{d–allh} = \text{MAX}_h\ \text{CT}(g,h,\text{CM}) + \Sigma_h\ \text{CT}(h,g,\text{XMLO}_{qih})/w.$$

Bibliography

[ANSI 76]
"ANSI/X3/SPARC Study Group on Database Management Systems Interim Report, 75-02-08," *FDT-Bulletin of the ACM SIGMOD*, 7, 2, June 1976, pp. 97–137.

[Appleton 78]
Appleton, D. S., "DDP Management Strategies: Keys to Success or Failure," *DATABASE*, Vol. 10, Number 1, Summer 1978, pp. 3–8.

[ARPA 76a]
ARPA, "Interface Message Processor: Specifications for the Interconnection of a Host and an IMP," Rept. No. 1822, Bolt Beranek and Newman, Inc., Cambridge, Massachusetts, January 1976.

[ARPA 76b]
ARPA, "ARPANET Protocol Handbook," NIC 7104, Network Information Center, Stanford Research Institute, Menlo Park, California, revised, April, 1976.

[Becker 78]
Becker, Hal B., "Let's Put Information Networks into Perspective," *Datamation*, March 1978, pp. 81–86.

[Bernstein 77]
Bernstein, Philip A., David W. Shipman, James B. Rothnie, and Nathan Goodman, "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The General Case)," Technical Report CCA-77-09, Computer Corporation of America, December 15, 1977.

[Buneman 76]
Buneman, O. Peter and E. K. Clemons, "Efficiently Monitoring Relational Databases," Decision Sciences Working Paper 76-10-08, *TODS* Vol. 4, Number 3, September 1979, pp. 368–382.

[Buneman 77b]
Buneman, O. Peter, Howard L. Morgan and Stanley F. Cohen, "Network Alerter Service: Preliminary Design," Decision Sciences Working Paper 77-07-03.

[Buneman 77b]
Buneman, O. Peter and H. L. Morgan, "Alerting Techniques in Database Systems," Decision Sciences Working Paper 77-03-04.

[Buneman 79]
Buneman, O. Peter and R. E. Frankel, "FQL—A Functional Query Language," *Proceedings SIGMOD 1979*, pp. 52–57.

[Carr 70]

Carr, C. S., S. D. Crocker, and V. G. Cerf, "Host/Host Communications Protocol in the ARPA Network," *Proceedings AFIPS Spring Joint Computer Conference*, May 1970.

[Chen 76]
Chen, Peter-Shen, "The Entity-Relationship Model—Toward a Unified View of Data," *TODS*, Vol. 1, No. 1, pp. 9–36.

[Chu 76]
Chu, Wesley W., "Performance of a File Directory System for Databases," *National Computer Conference 1976*, pp. 577–587.

[Clemons 76]
Clemons, Eric K., "Design of a User Interface for a Relational Database," Ph.D. Dissertation, Cornell University, 1976.

[Clemons 77]
Clemons, Eric K., "Design of an External Schema Facility to Support Database Update," Decision Science Working Paper 77-08-04.

[Clemons 78a]
Clemons, Eric K., "An External Schema Facility for CODASYL 1978," Decision Science Working Paper 78-10-03, *Proceedings VLDB 5*, Rio de Janeiro, Brazil, October 1979.

[Clemons 78b]
Clemons, Eric K., "Design of an External Schema Facility to define and Process Recursive Structures," Decision Science Working Paper 78-12-07, to appear in *TODS*.

[Clemons 79]
Clemons, Eric K. and Frank Germano, Jr., "An Experimental Implementation of an External Schema Facility for CODASYL," Decision Science Working Paper 79-03-02.

[CODASYL 71]
CODASYL, CODASYL Database Task Group April 1971 Report, available from ACM, New York.

[CODASYL 78]
CODASYL, "CODASYL Data Description Language Committee Journal of Development 1978", Material Data Management Branch, Dept. of Supply Services, 11 Laurier St., Hull, Quebec, Canada K1A 0S5.

[Cohen 77]
Cohen, Stanely F., "Alerting on Network Databases," Decision Sciences Working Paper 77-02-07, December 1976.

[Cortes 77]
Cortes, Ricardo, "An Alerting System for a Database Management System," Decision Sciences Working Paper 77-01-06, December 1976.

[Crocker 72]
Crocker, S. D., J. F. Heafner, R. M. Metcalfe, and J. B. Postel, "Function-oriented Protocols for the ARPA Computer Network," *Proceedings AFIPS Spring Joint Computer Conference*, May 1972.

[Date 76]
Date, C. J. "An Architecture for High-Level Language Database Extensions," *Proceedings SIGMOD Conference*, 1976.

[Date 77]
Date, C. J., *An Introduction to Database Systems*, 2nd Edition, Addison-Wesley, Menlo Park, California, 1977.

[Davenport 77]
Davenport, R. A. "Distributed or Centralised Database," *Computer Journal*, Vol. 21, No. 1.

[Einslow]
Einslow, Philip H., Jr., "What is a Distributed Data Processing System?," *Computer*, January 1978, pp. 13–21.

[Elam 77]
Elam, Joyce and J. Stutz, "Considerations for the Design of a Distributed Database System," accepted ACM National Conference, October, 1977.

[Elam 78]
Elam, Joyce, "A Model for Distributing a Database," Decision Sciences Working Paper 78-12-03, March 1978.

[Epstein 78]
Epstein, R., M. Stonebraker, and E. Wong, "Distributed Query Processing in a Relational Database System," *Proceedings SIGMOD 1978.*

[Fisher 78]
Fisher, Marshall L., "Lagrangian Relaxation Methods for Combinatorial Optimization," Decision Science Working Paper 78-10-06.

[Friedman 77]
Friedman, Frank and Elliott Koffman, *Introduction to Problem Solving and Structured Fortran*, Addison Welsey, Menlo Park, California, 1977.

[Fry 72]
Fry, J. P., D. P. Smith, and R. W. Taylor, "An Approach to Stored Data Definition and Translation," *ACM 1972 SIGFIDET Workshop on Data Description, Access and Control*, Denver, pp. 13–55.

[Fry 77]
Fry, James P. and John Mauer, "Operational and Technological Issues in Distributed Databases," Auerbach Publishers, 1977.

[Fry 78]
Fry, James P. and D. Swarthout, "Towards the Support of Integrated Views of Multiple Databases: An Aggregate Schema Facility," *Proceedings SIGMOD 1978.*

[Gambino 77]
Gambino, Thomas J. and Rob Gerritsen, "A Database Design Decision Support System," *Third Annual VLDB Conference*, Tokyo, Japan, 1977.

[Garcia-Molina 79]
Garcia-Molina, Hector, "A Concurrency Control Mechanism for Distributed Databases which Uses Centralized Locking Controllers," *Proceedings of the Fourth Berkeley Workshop on Distributed Computing*, August 1979, pp. 113–124.

[Garfinkel 72]
Garfinkel, Robert S. and George L. Nemhauser, *Integer Programming*, John Wiley and Sons, Inc., New York, 1972.

[Geller 75]
Geller, Dennis P., "The Principle of Sufficient Reason: A Guide to Language Design for Parallel Processing," *Proceedings SIGPLAN Conference*, March 1975.

[Germano 75]
Germano, Frank Jr. and Stephen Weyl, "A Database Organization to Support a Time-Oriented Medical Record," *Proceedings ACM Pacific 75*, San Francisco, April 1975.

[Germano 76]
Germano, Frank Jr., "The Design of a Mini-Based Time-Oriented Medical Databank System," *Proceedings of 1976 MUMPS Users' Group*, Madison, September 1976.

[Germano 78a]
Germano, Frank Jr. and Rob Gerritsen, "Distributed DBMS Software Architectures," Decision Sciences Working Paper 78-05-03, Wharton School, University of Pennsylvania, Philadelphia, May 1978.

[Germano 78b]
Germano, Frank Jr. and Meru Thakur, "SEEDFE: A For Each Data Language for SEED," Decision Sciences Working Paper 78-12-05.

[Gerritsen 75a]
Gerritsen, Rob, "A Preliminary System for the Design of DBTG Data Structures," *CACM* Vol. 18, No. 10, October 1975, pp. 551–557.

[Gerritsen 75b]
Gerritsen, Rob, "HI-IQ (Hierarchical Interactive Query): An Implemented Query Language for DBTG Databases," Department of Decision Sciences Working Paper 76-09-03, Wharton School, University of Pennsylvania, Philadelphia, September 1976.

[Gerritsen 76]
Gerritsen, Rob and Howard L. Morgan, "Dynamic Restructuring of Databases with Generation Data Structures," *Proceedings 1976 ACM National Conference.*

[Gerritsen 77a]
Gerritsen, Rob, Thomas J. Gambino, and Frank Germano, Jr., "Cost Effective Database Design: An Integrated Model," Decision Sciences Working Paper 77-12-03, Department of Decision Sciences, Wharton School, University of Pennsylvania, Philadelphia.

[Gerritsen 77b]
Gerritsen, Rob, Jack Buchanan, and David Root, "Automated Database Programming," Decision Sciences Working Paper 77-07-05, 1977 to appear in *CACM.*

[Gerritsen 78]
Gerritsen, Rob and Eric K. Clemons, "Self-Managing Database Systems Decision Sciences Working Paper 78-03-05, Department of Decision Sciences, Wharton School, University of Pennsylvania, Philadelphia, March 1978.

[Grapa 77]
Grapa, Enrique and Geneva G. Belford, "Some Theorems to Aid in Solving the File Allocation Problem," *CACM*, Vol, 20, No. 11, November 1977, pp. 878–882.

[Hayward 78]
Hayward, Jonathan, Rajeev Sangal, and Peter Buneman, "Q—A Communication Query Language for SEED," Decision Sciences Working Paper 78-05-02, May 1978.

[Hearst 70]
Hearst, F. E., W. R. Crowther, R. E. Kohn, S. M. Ornstein, and D. C. Walden, "The Interface Message Processor for the ARPA Computer Network," *Proceedings AFIPS Spring Joint Computer Conference*, May 1970.

[Hochbaum 78]
Hochbaum, Dorit S. and Marshall Fisher, "Database Location in Computer Networks," Decision Science Working Paper 78-10-09.

[Housel 76]
Housel, B. C. and N. C. Shu, "A High-Level Data Manipulation Language for Hierarchical Data Structures," *Proceedings of Conference on Data Abstraction, Definition and Structure*, Salt Lake City, March 1976, pp. 155–168.

[Housel 75]
Housel, B. C., D. P. Smith, N. C. Shu, and V. Y. Lum, "DEFINE—A Nonprocedural Data Description Language for Defining Information Easily," *Proceedings of ACM Pacific 75*, San Francisco, April 1975, pp. 62–70.

[IBM 73]
IBM, "Information Management System 360, Version 2: Application Programming Reference Manual," SH20-0912-4, 1973.

[Intel 78]
Intel, "Product Proposal: A Database Computer," Intel Memory Systems, Intel Corporation, Sunnyvale, California 94086.

[Jaikaumar 78]
Jaikaumar, Ramchandran and Marshall Fisher, "A Decomposition Algorithm for Large Scale Vehicle Routing," Decision Science Working Paper 78-11-05.

[Kaplan 79]
Kaplan, Jerrold, "Cooperative Responses from a Portable Natural Language Data Base Query System," Ph.D. Dissertation, University of Pennsylvania, Philadelphia, 1979.

[Kaufman 78]
Kaufman, Felix, "Distributed Process—A Discussion for Executives Traveling over Difficult EDP Terrain," *DATABASE*, Vol. 10, No. 1, Summer 1978, pp. 9–13.

[Kleinrock 74]
Kleinrock, Leonard and William E. Naylor, "On Measured Behavior of the ARPA Network," *Proc. NCC 1974*, Vol. 43, May 6–10, 1974, pp. 767–780.

[Lamport 77]
Lamport, Leslie, "Concurrent Reading and Writing," *CACM*, Vol. 20, No. 11, November 1977, pp. 806–811.

[Levin 74]
Levin, K. Dan, "Organizing Distributed Databases in Computer Networks," Decision Sciences Technical Report, Wharton School, University of Pennsylvania, Philadelphia, September 1974.

[Levin 75]
Levin, K. Dan and Howard Lee Morgan, "Optimizing Distributed Databases—A Framework for Research," *Proceedings NCC 1975*, pp. 473–478.

[Lien 78]
Lien, Y. Edmund and Peter J. Weinberger, "Consistency, Concurrency, and Crash Recovery," *Proceedings SIGMOD 1978*, pp. 9–14.

[Lin 79]
Lin, Wen-Te K., "Concurrency Control in a Multiple Copy Distributed Database System," *Proceedings of the Fourth Berkeley Workshop On Distributing Computing*, August 1979, pp. 207–220.

[Mamrak 77]
Mamrak, Sandra A., "Dynamic Response Time Prediction for Computer Networks," *CACM*, Vol. 20, No. 7, July 1977, pp. 461–468.

[Marci 76]
Marci, Philip P., "Deadlock Detection and Resolution in a CODASYL-Based Data Management System," *Proceedings SIGMOD 1976*, pp. 45–49.

[Martin 75]
Martin, James, *Computer Data-Base Organization*, 2nd Edition, Prentice-Hall, Englewood Cliffs, New Jersey, 1975.

[Maryanski 76]
Maryanski, Fred J., Paul S. Fisher, and Virgil E. Wallentine, "A User-Transport Mechanism for the Distribution of a CODASYL Data Management System," Kansas State University Technical Report TR CS 76-22, December 1976.

[Maryanski 77]
Maryanski, Fred J., and Paul S. Fisher, "Rollback and Recovery in Distributed Database Management Systems," *Proceedings 1977 ACM Annual Conference*, Seattle.

[Menasce 78]
Menasce, D. A., G. J. Popek, and R. R. Muntz, "A Locking Protocol for Resource Coordination in Distributed Databases," *Proceedings SIGMOD 1978*.

[Minoura 79]
Minoura, Toshimi, "A New Concurrency Control Algorithm for Distributed Databases," *Proceedings of the Fourth Berkeley Workshop on Distributed Computing*, August 1979, pp. 221–234.

[Morgan 77]
Morgan, Howard Lee and K. Dan Levin, "Optimal Program and Data Locations in Computer Networks," *CACM* Vol. 20, No. 5, May 1977, pp. 315–322.

[Morris 77]
Morris, P. and D. Sagalowiczs, "Managing Network Access To A Distributed Database," *Proceedings of the Second Berkeley Workshop on Distributed Data Management and Computer Networks*, May 1977.

[Peebles 73]
Peebles, Richard W., "Design Considerations for a Distributed Data Access System," Ph.D. Dissertation, University of Pennsylvania, Philadelphia, May 1973.

[Peebles 75]
Peebles, Richard W. and Eric Manning, "A Computer Architecture for Large (Distributed) Databases," *Proceedings VLDB 1975*, pp. 405–427.

[Prenner 77]
Prenner, C. J., "A Uniform Notation for Expressing Queries," Electronic Research Lab UCB/ERL M77/60, College of Engineering, University of California, Berkeley, September 1977.

[Ramirez 73]
Ramirez, J. A., "Automatic Generation of Data Conversion Programs Using a Data Description Language (DDL)," Vols. 1 and 2, University of Pennsylvania, Philadelphia, May 1973.

[Ramirez 74]
Ramirez, J. A., N. A. Rin, and N. S. Prywes, "Automatic Generation of Data Conversion Programs using A Data Description Language," *Proceedings, ACM SIGMOD Workshop on Data Description, Access and Control*, Ann Arbor, Michigan, May 1974, pp. 207–225.

[Ries 79a]
Ries, D. and Michael R. Stonebraker, "Locking Granularity Revisited," *ACM TODS*, Vol. 4, No. 2, June 1979, pp. 210–227.

[Ries 79b]
Ries, D., "The Effects of Concurrency Control on the Performance of a Distributed Data Management System," *Proceedings of the Fourth Berkeley Workshop on Distributed Data Management*, August 1979, pp. 75–112.

[Rosenkrantz 78]
Rosenkrantz, Daniel J., Richard E. Stearns, and Philip M. Lewis II, "System Level Concurrency Control for Distributed Database Systems," *ACM Transactions of Database Systems*, Vol. 3 No. 2, June 1978, pp. 178–198.

[Rothnie 77a]
Rothnie, J. B. and N. Goodman, "An Approach to Updating in a Redundant Distributed Database Environment," Computer Corporation of America TR CCA-77-01, February 15, 1977.

[Rothnie 77b]
Rothnie, James B. and Nathan Goodman, "An Overview of the Preliminary Design of SDD-1: A System for Distributed Databases," *Proceedings of the Second Berkeley Workshop on Distributed Data Management and Computer Networks*, May 1977, pp. 39–57.

[Schmidt 77]
Schmidt, Joachim W., "Some High Level Language Constructs for Data of Type Relation," *ACM Transactions on Database Systems*, Vol. 1, No. 3, September 1977, pp. 247–261.

[Schneider 77]
Schneider, L. S., "A Relational Query Compiler for Distributed Heterogeneous Databases," unpublished paper, 1977.

[Schwager 78]
Schwager, Andre O., "The Hewlett-Packard Distributed System Network," Hewlett-Packard Systems J., 1978.

[SEED 77]
SEED Reference Manual, International Database Systems, Inc., Philadelphia, 1977.

[Shipman 79]
Shipman, David W., "The Functional Data Model and the Data Language DAPLEX," *Supplement to the Proceedings SIGMOD 1979*, pp. 1–19.

[Shu 77]
Shu, N. C., B. C. Housel, R. W. Taylor, G. P. Grosh, and V. Y. Lum, "EXPRESS: A Data EXtraction, Processing, and REStructuring System," *TODS*, Vol. 2, No. 2, June 1977, pp. 134–174.

[Sibley 73]
Sibley, E. H. and R. W. Taylor, "A Data Definition and Mapping Language," *CACM* Vol. 16, No. 12, December 1973, pp. 750–759.

[Sibley 76]
Sibley, E. H. (Ed), "Special Issue: Database Management Systems," *ACM Computing Surveys 8*, 1, March 1976.

[Stemple 76]
Stemple, David W., "A Database Management Facility for Automatic Generation of Database Managers," *TODS*, Vol. 1, No. 1, March 1976, pp. 79–94.

[Stonebraker 76]
Stonebraker, Michael, "Proposal for a Network INGRESS," *Proceedings of the Berkeley Workshop on Distributed Data Management and Computer Networks*, May 1976.

[Stonebraker 77]
Stonebraker, Michael and Erich Neuhold, "A Distributed Database Version of Ingress," *Proceedings of the Second Berkeley Workshop on Distributed Data Management and Computer Networks*, May 1977, pp. 39–57.

[Stonebraker 79a]
Stonebraker, Michael, "The Argument Against CODASYL," *Proceedings ACM-SIGMOD 1979, Supplement*, Boston, May 1979, pp. 58–62.

[Stonebraker 79b]
Stonebraker, Michael, "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," *IEEE Transactions of Software Engineering*, Vol. 4, SE-5, No. 3, May 1979, pp. 188–194.

[Thomas 76]
Thomas, Robert H., Stuart C. Schaffner, "MSG: The Interprocess Communication Facility for the National Software Works," BBN Report No. 3483, December 1976.

[Weyl 75]
Weyl, Stephen, James Fries, Gio Wiederhold, and Frank Germano, Jr., "A Modular, Self-Describing Clinical Databank System," *Computers and BioMedical Research*, Vol. 8, 1975, pp. 279–293.

Index

ISBN 0-8357-1221-4